Syntax-basiertes Program-Slicing von COBOL-Programmen**

(IFI TR 95.30)

Zafer Öztürk *

Institut für Informatik der Universität Zürich,
Winterthurerstr. 190, 8057 Zürich, Schweiz

Email: oeztuerk@ifi.unizh.ch
Telefon: +41-1-257 4307. Fax: +41-1-363 0035.

Stichworte: Program-Slicing, COBOL

Inhalt: Der Begriff des Program-Slice wurde durch M. Weiser [Wei82, Wei84] geprägt. Nach seiner Auffassung zerlegen Wartungsprogrammierer das zu wartende Programm mental in kleinere semantisch verbundene Teile. Das Program-Slicing ist darum bestens geeignet, um das Program-Understanding in der Wartung zu unterstützen.

Am Institut für Informatik an der Universität Zürich wird gegenwärtig im Rahmen des Projekts AEMES[1] eine erweiterbare Wartungsumgebung für COBOL-Applikationen entwickelt und prototypisch implementiert. Wir werden mehrere Ansätze für das syntax-basierte Program-Slicing vorstellen, die ausschliesslich gestützt auf Informationen aus den Cross-Referencer-Daten operieren. Dadurch lassen sich unsere Ansätze auch da einsetzen, wo die interne Datenrepräsentation aus der Übersetzungsphase des Compilers für das Program-Slicing nicht zur Verfügung steht.

**Der vorliegende Bericht erscheint in identischer Form im Dezember '95 auch als Bericht 13/95 der Informatik-Forschungsgruppe der Lebensversicherungs- und Rentenanstalt/Swiss Life.

**Zafer Öztürk* wird unterstützt durch die Schweizerische Lebensversicherungs- und Rentenanstalt/Swiss Life.

[1] AEMES steht für "**A**n **E**xtensible **M**aintenance **E**ngineering **S**ystem". Es ist ein Projekt, das zum Ziel hat, dem Wartungsprogrammierer eine Sammlung von Werkzeugen in einer erweiterbaren Umgebung anzubieten. Dieses Projekt wird unterstützt durch das KWF (Kommission zur Förderung der Wissenschaftlichen Forschung), die Schweizerische Lebensversicherungs- und Rentenanstalt/Swiss Life und Bull(Schweiz).

Inhaltsverzeichnis

1 Einleitung **3**

1.1 Das Projekt AEMES . . . 3

1.2 Program-Slicing . . . 4

2 Vorbemerkungen **6**

2.1 Auswirkungen der zeilen-orientierten Betrachtung des Programmtexts . . . 6

2.2 Regeln für den Aufbau des Programmtexts . . . 9

3 Verwendete Grössen und Hilfsfunktionen **11**

3.1 Beschreibung der in den Ansätzen verwendeten Grössen . . . 11

3.2 Hilfsfunktionen . . . 12

3.2.1 Basisfunktionen . . . 12

3.2.2 Zugriffsfunktionen auf Cross-Referencer-Daten . . . 13

3.2.3 Funktionen zur Generierung von Strukturdaten . . . 13

3.2.4 Funktionen zum Sammeln von einfachen Programmobjekten . . . 13

4 Erster Lösungsansatz **14**

4.1 Program-Slicing anwenden auf den gesamten Programmtext . . . 14

4.1.1 Kurzbeschreibung . . . 14

4.1.2 Beschreibung der Funktionen . . . 15

4.2 Beschränken des Program-Slicing auf einen Paragraphen . . . 18

4.2.1 Kurzbeschreibung . . . 18

4.2.2 Beschreibung der Funktionen . . . 19

5 Verbesserter Lösungsansatz **22**

5.1 Begriffe und Vorgehen . . . 22

5.1.1 Aufspalten des Betrachtungsbereichs . . . 24

5.1.2 Sequenz der Untersuchungsbereiche für rückwärts-orientierte Slices 25

5.2 Beschreibung der Funktionen . . . 27

6 Evaluation **35**

6.1 Eigenschaften der syntax-basierten Slices . . . 35

6.1.1 Die ersten beiden Ansätze . . . 35

6.1.2 Der verbesserte Ansatz . . . 36

6.2 Diskussion . . . 37

7 Zusammenfassung und Ausblick **39**

1 Einleitung

1.1 Das Projekt AEMES

Viele Firmen stehen heute vor dem Problem, z.T. sehr alte Applikationen warten zu müssen und gegebenenfalls an neue Umgebungen anzupassen. Diese an sich schon aufwendige Arbeit wird noch zusätzlich dadurch erschwert, dass zum einen die vorhandenen Dokumentationen in aller Regel nicht den aktuellen Stand der Programme widerspiegeln, und zum anderen auch die vormals zuständigen Entwickler nicht mehr verfügbar sind. Viele dieser Applikationen enthalten implizites, d.h. hoch-spezialisiertes Wissen über komplexe Geschäftsabläufe einer Firma und erfüllen in zentralen Bereichen wichtige Aufgaben. Eine Ablösung der mit hohen Investitionskosten belasteten alten Applikationen durch neue Produkte, welche wiederum beträchtliche Mittel erfordern, wird aber meist aus betriebs-ökonomischen Überlegungen hinausgezögert. Die Firmen sind also gezwungen mit solchen überalterten Applikationen weiterzuarbeiten und sie irgendwie am Leben zu erhalten. Es ist darum für die Wartungspraxis sehr wichtig, ein Instrument zu entwickeln, welches das Verstehen der Funktionalität (Program-Understanding) von alten, mangelhaft dokumentierten Applikationen so einfach und effizient wie möglich macht.

An der Universität Zürich wird im Rahmen des Projekts AEMES ein Prototyp einer erweiterbaren Umgebung für das Reverse Engineering von COBOL-Programmen entwickelt ([FKO92, BFK+93, BFKO93a, BFKO93b, BFK+94]). Die praxis-orientierte Zielsetzung ist, den Wartungsprogrammierer in seiner täglichen Arbeit best möglich zu unterstützen. Dabei konzentriert sich die projekt-spezifische Entwicklungsarbeit von AEMES auf die folgenden drei Hauptbereiche:

1. Wie können COBOL-Programmtexte in geeigneter Form intern repräsentiert werden, und wie lässt sich ein bestehender Programmtext in diese Darstellung überführen?

2. Welche zusätzlichen Informationen können aus der gewonnenen internen Darstellung generiert werden, die den Wartungsprogrammierer beim Program-Understanding unterstützen?

3. Wie können diese Informationen in einer problemgerechten und vor allem benutzerfreundlichen Weise dargestellt werden, ohne den Wartungsprogrammierer mit Daten zu überfluten?

Für die Transformation des Programmtexts in eine geeignete interne Repräsentation verweisen wir auf [Fae95]. Was die Präsentation der daraus generierten Informationen betrifft, wurde in [Kis94] das Konzept des *Wireable User Interface* vorgeschlagen.

Unser Interesse gilt vor allem dem zweiten Hauptbereich, wo es darum geht, Ansätze zu entwickeln, die aus der internen Repräsentation des Programmtexts weitere, für den Wartungsprogrammierer nützliche Informationen gewinnen. Das Spektrum solcher Werkzeuge ist weit gefächert; sie reichen von Pretty-Printern und Cross-Referencern, über Daten- und Kontrollfluss-Analysatoren, bis hin zu Plan-Generatoren.

Im vorliegenden Bericht werden wir aus diesem Spektrum das Program-Slicing betrachten, welches uns ermöglicht (mit Hilfe von Daten- und Kontroll-Abhängigkeiten), jene Teile aus dem Programmtext zu bestimmen, die auf eine interessierende Programmstelle einen Einfluss ausüben können.

1.2 Program-Slicing

Der Begriff des Program-Slicing wurde durch Mark Weiser geprägt ([Wei82, Wei84]). Er vertritt die Auffassung, dass Wartungsprogrammierer das zu wartende Programm mental in kleinere semantisch verbundene Teile zerlegen. In [Wei84] wird auch ein Algorithmus beschrieben, wie diese Slices berechnet werden können. In der Folge wurden weitere Ansätze für das Program-Slicing entwickelt (z.B. [KL88, Hau89, HRB90, Oez95]). Die Program-Slices lassen sich mit folgenden drei Kriterien-Paaren klassifizieren (die daraus entstehenden Kombinationen und ihre Eigenschaften sind in [Ven91] dargestellt):

1. statisch $\longleftrightarrow$ dynamisch

2. vorwärts $\longleftrightarrow$ rückwärts

3. ausführbar $\longleftrightarrow$ nicht-ausführbar

Bisherige Ansätze für das Program-Slicing stützen sich auf die vollständige Information aus dem Parsbaum, woraus dann eine geeignete, interne Repräsentation der Kontroll- und Datenabhängigkeiten (z.B. *Program Dependence Graph*) generiert wird ([OO84, FOW87]). Es gibt allerdings auch Ansätze, in denen vorausgesetzt wird, dass die Semantik der untersuchten Programmiersprache formal definiert vorliegt ([Hau89, Oez95]).

Nun heisst aber im Projekt AEMES die Zielsprache COBOL, was nicht nur, aber auch im Hinblick auf das Program-Slicing einige zusätzliche Schwierigkeiten mit sich bringt.

Abgesehen vom enormen Sprachumfang (ca. 450 Schlüsselwörter) und den diversen anbieterspezifischen Erweiterungen, ist die Beschreibung der einzelnen Sprachkonstrukte von COBOL stellenweise ungenau, manchmal sogar widersprüchlich. Es kann deshalb nicht überraschen, dass sich die Semantik der Sprachkonstrukte nicht eindeutig festlegen lässt, was aber beispielsweise für die oben erwähnten Ansätze in [Hau89] und [Oez95] grundlegende Voraussetzung wäre.

Erwähnenswert ist der Ansatz eines Parser-Generators in [Fae95], der an Hand einer Grammatik in EBNF-Notation eines COBOL-Dialekts den Parser generieren kann. Dieser Parser erzeugt den Parsbaum und stellt diesen für weitere Werkzeuge zur Verfügung. In einer kommerziellen Umgebung findet man aber nach unserer Kenntnis ausschliesslich COBOL-Compiler, deren intern erzeugte Zwischenrepräsentationen nicht zugänglich sind. Eine Option, die aber von fast allen COBOL-Compilern angeboten wird, besteht im Verwendungsnachweis der Programmobjekte (z.B. Felder, Paragraphen, ...). Zumindest diese Liste steht in der Regel zur Verfügung (cross references).

Aus diesem Grund beschränken wir uns bewusst nur auf solche Informationen, welche die meisten COBOL-Compiler dem Benutzer zur Verfügung stellen. Wir werden untersuchen, ob und in welchem Rahmen das Program-Slicing von *real-world Applikationen* in COBOL möglich ist, welches sich ausschliesslich auf allgemein zugängliche Daten stützt.

Der vorliegende Bericht ist wie folgt aufgebaut: In Abschnitt 2 werden die durch ausschliessliche Verwendung von Cross-Referencer-Daten möglichen Effekte aufgezeigt. Im Sinne einer Empfehlung werden zudem einige Regeln zur Gestaltung des Programmtexts formuliert, womit die Voraussetzungen für das Program-Slicing optimiert werden können. Anschliessend werden im 3. Abschnitt Grössen und Hilfsfunktionen eingeführt und beschrieben, welche in den von uns vorgestellten Ansätzen zur Anwendung kommen. In Abschnitt 4 werden zunächst zwei Ansätze ausgearbeitet, welche richtungs-unabhängige Program-Slices generieren. In Abschnitt 5 wird sodann unter Berücksichtigung der Analyserichtung der Umfang der Slices weiter reduziert. Die aus den drei Ansätzen gewonnenen Resultate werden wir in Abschnitt 6 besprechen und diesen Bericht mit Zusammenfassung und Ausblick in Abschnitt 7 beschliessen.

2 Vorbemerkungen

2.1 Auswirkungen der zeilen-orientierten Betrachtung des Programmtexts

In der Einleitung haben wir erwähnt, dass wir uns im vorliegenden Ansatz des Program-Slicing auf die Informationen, welche jeder gängige COBOL-Compiler (oder separate Cross-Referencer) liefern kann, beschränken wollen. In diesem Abschnitt soll gezeigt werden, welche Auswirkungen diese Einschränkung nach sich zieht.

Was aus dem Cross-Referencer an Informationen zur Verfügung steht, sind zum einen die Objekte (z.B. Variablennamen, Schlüsselworte, ...) und zum anderen die zugehörigen Programmzeilen, in welchen sie verwendet werden. Diese rudimentären Angaben lassen jedoch eine exakte Lokalisierung der Objekte innerhalb der Programmzeile nicht zu. Wäre diese Information verfügbar, so liesse sich beispielsweise aus der Anweisung "MOVE `a` TO `x`" leicht ableiten, dass die Variable vor dem Schlüsselwort "TO" die gelesene und die Variable nach dem Schlüsselwort die modifizierte Variable ist. Nun aber ist in unserem Fall die Verwendung selbst so einfacher Heuristiken nicht möglich, weil ein üblicher Cross-Referencer lediglich die Zeilennummern liefert, nicht aber die erforderlichen Spaltennummern.

Relationen zwischen Variablen

Die erste grosse Einschränkung betrifft die Beziehungen der Variablen untereinander: Da sich aufgrund der fehlenden Lokalisierbarkeit der Objekte innerhalb der Zeile nicht bestimmen lässt, welchen der verwendeten Variablen nur eine passive Rolle zukommt, müssen wir, um sicher zu gehen, annehmen, dass jede Variable einer Programmzeile von allen anderen Variablen der gleichen Zeile sowohl gelesen als auch modifiziert werden kann. Daraus folgt direkt, dass sämtliche Variablen einer Zeile als gegenseitig abhängig betrachtet werden müssen.

Von dieser Einschränkung wird, wie wir gleich sehen werden, auch das Slicing-Kriterium betroffen. Das Kriterium besteht in der Regel aus zwei Komponenten: zum einen aus der Programmstelle und zum anderen aus der Variablen, für die das Program-Slicing durchgeführt wird. In unserem Fall aber, wo alle in der gleichen Zeile vorkommenden Variablen als voneinander abhängig betrachtet werden müssen, ist unschwer abzuleiten, dass auch für jede dieser Variablen identische Program-Slices resultieren. Aus dieser Erkenntnis erübrigt sich die Angabe eines bestimmten Variablennamens, da in jedem Fall alle Variablennamen auf der selben Zeile gleichzeitig betrachtet werden. Das Slicing-Kriterium in unserem Ansatz besteht demnach nur noch aus der Angabe der Zeilennummer der zu untersuchenden Programmstelle.

Umgebung einer Variablen

Die zweite gravierende Einschränkung betrifft die zusammengesetzten Sprachkonstrukte: Die Information über die Zugehörigkeit zu einer bestimmten Umgebung (Scope) geht verloren. Mit der Zeilennummer einer Anweisung allein ist nicht zu bestimmen, ob sie sich innerhalb einer übergeordneten Struktur (z.B. bedingte Anweisung, Paragraph, ...) befindet und gegebenenfalls in welcher Schachtelungstiefe. Um solche Informationen aus den Angaben eines Cross-Referencers ableiten zu können, ist es erforderlich, dass auf die Eigenschaften gewisser Anweisungen zurückgegriffen wird.

Betrachten wir als Beispiel die bedingte Anweisung in Abbildung 1. Sie besteht aus zwei Sequenzen von Anweisungen. Abhängig vom Wert der Bedingung `(c < 5)` wird die eine oder die andere ausgeführt.

```
IF (c < 5)
THEN
   MOVE a TO x
ELSE
   MOVE b TO x
```

Abbildung 1: Programmbeispiel zum Gültigkeitsbereich

Nehmen wir an, dass die Anweisung `MOVE a TO x` in das Slice übernommen werden soll. Diese Anweisung ist zwar, isoliert betrachtet, syntaktisch korrekt, und auch das neue Slice wird es sein; aber die Semantik dieses Programmteiles wäre im Slice nicht mehr identisch mit dem ursprünglichen Programm. Darum muss bei der Übernahme einer Anweisung in das Slice, auch der sie umgebende Kontext betrachtet werden. Ähnliches gilt auch für die Sprachkonstrukte GO TO und PERFORM mit all ihren Variationen.

Den Einbezug der Umgebung (scope) einer Anweisung werden wir *Syntaktisches Ergänzen* nennen. Diese wird immer dann erforderlich, wenn Fragmente aus Sprachkonstrukten in das Slice übernommen wurden.

Aus diesen Ausführungen ergibt sich als Konsequenz, dass die Struktur von gewissen Sprachkonstrukten (bedingte Anweisung, Sprünge, Paragraphen, ...) in die rein syntaktische Generierung der Program-Slices miteinbezogen werden muss. Zur Repräsentation dieser Strukturen verwenden wir Paare resp. Tripel von Zeilennummern (siehe Abschnitt 3.1). Diese markieren Anfang und Ende der jeweiligen Struktur. Falls nun Anweisungen innerhalb einer solchen Struktur in das Program-Slice übernommen werden sollen, so müssen auch die Anfangs- und Endzeilen der Struktur in das Slice übernommen werden.

Dabei ist allerdings Vorsicht geboten. Es muss sicher gestellt sein, dass durch dieses Integrieren der Intervallgrenzen einer Struktur nicht zusätzliche Abhängigkeiten in das Slice eingeschleppt werden. Steht beispielsweise eine Anweisung, welche noch nicht in das Slice übernommen wurde, in der gleichen Zeile wie die Endzeile einer Struktur, so würde es so erscheinen, als ob die Variablen in dieser Anweisung auch einen Einfluss auf die Programmzeile des Slicing-Kriteriums hätten, was aber nicht zwingend der Fall sein muss. Es ergäben sich dadurch zwar keine fehlerhaften Slices, aber sie wären umfangreicher als erforderlich.

Aus diesem Grunde werden die in das Slice aufgenommenen Endzeilen von Strukturen speziell behandelt. Konkret: Je nach dem, ob die Endzeile eine Anweisung enthält, die in das Slice gehört oder nicht, wird die Endzeile als reguläre Zeile integriert oder aber nur als "Endmarke" mitgeführt (siehe Abschnitt 3.1).

Kontrollfluss des Programms

Eine weitere empfindliche Einschränkung, die sich bei einer ausschliesslichen Verwendung von Cross-Referencer-Informationen ergibt, besteht darin, dass der Kontrollfluss eines Programms nicht verfolgt werden kann. Aufgrund der oben aufgezeigten Schwierigkeiten bezüglich der Kontextsicherung von Anweisungen müssen darum alle Sprunganweisungen und Subroutinenaufrufe in das Program-Slice aufgenommen werden.

```
(1)  Para1.
(2)      MOVE c TO d.
(3)      PERFORM Para2.
(4)      MOVE b TO x.
(5)      MOVE d TO y.
(6)      GOTO Para3.
(7)  Para2.
(8)      MOVE a TO x.
(9)      MOVE 3 TO d.
(10) Para3.
(11)     EXIT.
```

Abbildung 2: Kontrollfluss in Program-Slice integrieren

Betrachten wir hierzu den Kontrollfluss in Abbildung 2 als Beispiel. Die Reihenfolge der durchlaufenen Zuweisungen lautet (2) → (8) → (9) → (4) → (5) → (11). Ein Blick auf den Wert der Variablen `x` am Ende des Programms zeigt, dass dieser in der Anweisung (4) letztmals beschrieben wird und darum den Wert von `b` erhält. In das Program-Slice für die Zeile (4) werden die relevanten Anweisungen (4) und (8) aufgenommen. Wird nun aber der Kontrollfluss bei der Berechnung des Slices vernachlässigt, so wird der Paragraph `Para2` nicht übersprungen, und die Variable `x` erhält den Wert der Variablen `a` aus der Anweisung (8) zugewiesen. Um solche Fehler zu vermeiden, wird grundsätzlich jede Sprunganweisung und jeder Subroutinenaufruf in das Program-Slice aufgenommen. Ähnliches gilt übrigens auch für die Variable `y`, die unter Missachtung des Kontrollflusses den Wert der Variablen `c` enthält, im Originalprogramm aber den konstanten Wert `3` aufweist.

2.2 Regeln für den Aufbau des Programmtexts

Da die Informationen aus dem Cross-Referencer grundsätzlich zeilen-orientiert sind, sollten bezüglich der Gestaltung eines Programmtexts einige Restriktionen beachtet werden, damit die Qualität des berechneten Program-Slices optimal ist. Im folgenden werden wir Anregungen und Empfehlungen zu einzelnen Sprachkonstrukten in COBOL geben, bei deren Beachtung die Berechnungen des Program-Slices unterstützt werden können.

Regel 1 (Einfache Anweisungen) *Jede einfache Anweisung (z.B.* MOVE, ADD, COMPUTE*) steht gesondert auf einer Zeile.*

Betrachten wir dazu das Beispiel in Abbildung 3: Hier fehlt uns die Kenntnis darüber, welche Variable von welchen beeinflusst wird, denn gemäss den zwingenden Einschränkungen unseres Ansatzes (Abschnitt 2.1) werden hier alle in einer Zeile auftretenden Variablen als voneinander abhängig betrachtet.

```
(a)    MOVE a TO x
       ADD  b TO y

(b)    MOVE a TO x ADD b TO y
```

Abbildung 3: Einfache Anweisungen

Im ersten Fall, wo die beiden einfachen Anweisungen auf separaten Zeilen stehen (Regel 1), wird beim Slicing angenommen, dass `a` und `x` resp. `b` und `y` in gegenseitiger Beziehung zueinander stehen. Im zweiten Fall würden gar alle Variablen `a`, `b`, `x` und `y` als in gegenseitiger Beziehung stehend betrachtet. Hier zeigt sich, dass zusätzliche Relationen zwischen Variablen geschaffen werden (z.B. zwischen `a` und `y`), welche das generierte Program-Slice unnötig aufblähen.

Regel 2 (Paragraphen- und Sectionsnamen) *Jeder Name eines Paragraphen resp. einer Section (gemeinsam mit dem Schlüsselwort* SECTION*) steht gesondert auf einer Zeile.*

Damit soll verhindert werden, dass bei der syntaktischen Ergänzung während der Berechnung eines Program-Slices überflüssige Anweisungen integriert werden, welche mit einem Paragraphennamen nur zufällig in der selben Zeile stehen.

Regel 3 (Bedingte Anweisungen) *Die Bedingung steht gemeinsam mit dem Schlüsselwort* IF *auf einer separaten Zeile. Falls das Schlüsselwort* ELSE *verwendet wird, so steht es ebenfalls auf einer gesonderten Zeile. Die verwendeten Anweisungssequenzen innerhalb einer bedingten Anweisung sind wiederum nach den Regeln zu gliedern.*

```
IF c>d
   MOVE a TO x
ELSE
   ADD  b TO y
```

Abbildung 4: Bedingte Anweisung

Eine nach der Regel 3 gruppierte bedingte Anweisung wäre darzustellen, wie in Abbildung 4 gezeigt. Hier stehen die Bedingung `c>d` und die beiden einfachen Anweisungen `MOVE a TO x` resp. `ADD b TO y` jeweils auf gesonderten Zeilen (Regel 1). Damit wird gewährleistet, dass die beiden Äste unabhängig voneinander betrachtet werden können, was z.B. dann erforderlich ist, wenn wir einen Ast, der keinen Beitrag für das Program-Slice liefert, vernachlässigen wollen.

Regel 4 (Schlüsselworte) *Es ist zu vermeiden, dass zwei identische Schlüsselworte auf der selben Zeile auftreten, da sonst durch den Cross-Referencer Informationen verloren gehen, weil er jedes Programm-Objekt pro Zeile nur einmal referenziert.*

Probleme, die auf ein "ungünstiges Layout" des Programmtexts zurückzuführen sind, lassen sich mittels eines Pretty-Printers leicht beheben oder zumindest entschärfen.

Das Einhalten der oben genannten Regeln ermöglicht dem hier vorgestellten Program-Slicing, präzisere Slices zu berechnen. Falls aber die Regeln nicht befolgt werden, so bedeutet dies nicht, dass falsche Ergebnisse resultieren, sondern nur *weniger genaue*, da grössere Slices zu erwarten sind.

3 Verwendete Grössen und Hilfsfunktionen

3.1 Beschreibung der in den Ansätzen verwendeten Grössen

Im folgenden Abschnitt werden die wichtigsten Grössen, welche in unserem Ansatz für das Program-Slicing Verwendung finden, aufgelistet und besprochen.

- **Slicing-Kriterium:** *int*

  Wir haben bereits weiter oben angedeutet, dass und weshalb im vorliegenden Ansatz der Variablenname als Bestandteil des Slicing-Kriteriums nicht benötigt wird. Durch die Verwendung der nur-zeilenorientierten Darstellung von Informationen aus dem Cross-Referencer fehlen uns die zur Lokalisierung der Variablen innerhalb der Zeile erforderlichen Spaltennummern. Es muss darum angenommen werden, dass alle in gegenseitiger Abhängigkeit stehen. Angesichts dessen ist es müssig, eine einzelne Variable betrachten zu wollen, denn alle Variablen auf der selben Programmzeile haben das identische Program-Slice!

- **Programmstrukturen:** $(int * int)$ resp. $(int * int * int)$

  Strukturen innerhalb eines Programmtexts (z.B. Paragraphen), werden als Paare von Zeilennummern repräsentiert, welche die erste resp. die letzte Zeile der zu betrachtenden Struktur bezeichnen. Die bedingte Anweisung, welche ebenfalls eine Programmstruktur darstellt, wird als Zahlentripel repräsentiert, wobei der mittlere Wert die Nummer der Zeile angibt, auf welcher das Schlüsselwort ELSE steht.

- **Slice:** $(int\ list * int\ list)$

  Ein Program-Slice umfasst all diejenigen Programmzeilen, die einen direkten oder indirekten Einfluss auf die Zeile des Slicing-Kriteriums haben. Es gibt zwei Modalitäten, unter denen eine Programmzeile in das Slice integriert werden kann:

  1. Die Programmzeile enthält Variablen, welche direkten oder indirekten Einfluss auf mindestens eine Variable in der Zeile des Slicing-Kriteriums haben. In diesem Falle wird sie in die *transitive Hülle* aufgenommen.
  2. Die Programmzeile ist Teil einer *syntaktischen Ergänzung*, d.h. sie gehört zu einem Programmkonstrukt (z.B. Endzeile einer bedingten Anweisung oder eines Paragraphen), welches Zeilen aus dem Program-Slice enthält. Diese Programmkonstrukte müssen ebenfalls in das Slice übernommen werden, damit der Kontext der Anweisungen auch innerhalb des Program-Slices erhalten bleibt.

Wir haben bereits im Abschnitt 2.1 angedeutet, dass diese syntaktische Ergänzung, insbesondere die Endzeilen von Programmkonstrukten, für das Program-Slicing zum Problem werden kann. Aus diesem Grund wird ein Program-Slice durch zwei Mengen von Zeilennummern dargestellt. Die erste Menge enthält die Nummern aller Zeilen (meist *slc* genannt), die vollständig im Slice benutzt werden. Eine zweite Menge umfasst die Nummern derjenigen Zeilen, die während einer *syntaktischen Ergänzung* als Endmarken in das Slice übernommen wurden (meist *dots* genannt), wobei aber die Anweisungen auf der selben Zeile für das Program-Slicing zu vernachlässigen sind.

3.2 Hilfsfunktionen

3.2.1 Basisfunktionen

`MIN`, `MAX`
: Diese Konstanten enthalten die kleinste resp. die grösste darstellbare positive Zahl. Sie werden zur Begrenzung der Zeilennummern verwendet.

`fill` : *(int * int) → int list*
: Funktion, die die Menge aller Zeilennummern liefert, welche innerhalb eines Intervalls liegen.
Bsp: fill (4,7) = [4,5,6,7]

`toSet` : *(int * int * ...) → int list*
: Diese Funktion liefert die Zeilennummern innerhalb eines n-Tupels als Menge zurück.
Bsp: toSet (2,4,5,5,5,6) = [2,4,5,6]

`max` : *'a list → 'a*
: Mit dieser Funktion wird das Maximal-Element einer Menge bestimmt. Falls 'a ein n-Tupel ist, entscheidet das erste Element.

`min` : *'a list → 'a*
: Analog zu `max` liefert diese Funktion das Minimal-Element einer Menge.

`apply` : *(int * ('a * 'b * 'c → 'b * 'c) * 'a * 'b * 'c) → 'b * 'c*
: Diese Funktion führt genau n-mal (erstes Argument) die übergebene Funktion (zweites Argument) aus, welche drei Parameter (drittes, viertes und fünftes Argument) verwendet.

`joinStates` : *((int list * int list) * (int list * int list)) → (int list * int list)*
: Mit dieser Funktion werden zwei Slices (erstes und zweites Argument) zu einem neuen Slice vereint. Jedes Slice wird dargestellt als ein Tupel aus Mengen von Zeilennummern.

`makePairs` : *(int * (int * int) list * int) → (int * int) list*
: Funktion zur Verkettung von Intervallen. Das erste und das letzte Argument markieren den Anfang resp. das Ende der Kette. Die Werte aus den Paaren des zweiten Arguments können als Stützpunkte für die Intervalle betrachtet werden.
Bsp: `makePairs` *(2, [(5,9), (17,24)], 28) = [(2,4), (5,9), (10,16), (17,24), (25,28)]*

3.2.2 Zugriffsfunktionen auf Cross-Referencer-Daten

`getLines` : *(int * int * string list) → int list*
: Mit dieser Funktion wird für eine Menge von Variablennamen (dritte Komponente) die Menge der entsprechenden Zeilennummern innerhalb eines Intervalls (erste und zweite Komponente) berechnet, in welchen mindestens eine der Variablen erscheint.

`getMembers` : *int list → string list*
: Eruiert für eine Menge von Zeilennummern die Menge der Variablennamen, die in mindestens einer der Zeilen auftreten.

`getMset` : *(string * int) → string list*
: Mit dieser Funktion wird die Menge der in einer bestimmten Zeile (zweite Komponente) vorkommenden Namen aus einer angegebenen Klasse (erste Komponente) errechnet.

`getLset` : *(int * int * string * string) → int list*
: Liefert die Menge der Zeilennummern innerhalb eines Intervalls (erste und zweite Komponente), in welchen ein bestimmter Name (vierte Komponente) aus einer Klasse (dritte Komponente) vorkommt.

3.2.3 Funktionen zur Generierung von Strukturdaten

`allTripleIfs` : *(int * int) → (int * int * int) list*
: Berechnet innerhalb des angegebenen Intervalls eine Menge von Tripeln, welche jedes eine bedingte Anweisungen darstellt. Die erste Komponente bezeichnet die Zeile, worin das Schlüsselwort IF auftritt. Die zweite Komponent bezeichnet die Zeile, worin das zugehörige ELSE vorkommt oder die Endzeile der bedingten Anweisung. Die letzte Komponente bezeichnet die Zeile, wo die bedingte Anweisung endet.

`allPairGos` : *(int * int) → (int * int) list*
: Liefert von allen Sprunganweisungen, die im angegebenen Intervall liegen, die Menge ihrer Anfangs- und Endzeilen als Paare.

`allPairPerfs`, `paraPairs`, `sectPairs`, `divPairs` : *(int * int) → (int * int) list*
: Entsprechend der Funktion `allPairGos`, liefern diese Funktionen für die Subroutinenaufrufe und die Strukturen der Paragraphen, Sections und Divisions, welche im angegebenen Intervall liegen, die entsprechende Menge ihrer Anfangs- und Endzeilen als Paare.

3.2.4 Funktionen zum Sammeln von einfachen Programmobjekten

`allVars` : *(int * int) → int list*
: Berechnet die Menge aller Zeilennummern im angegebenen Intervall, auf welchen mindestens eine Variable auftritt.

`allProcs` : *(int * int) → int list*
: Liefert die Menge von Zeilennummern innerhalb des angegebenen Intervalls, von welchen aus ein Paragraph aufgerufen wird.

`allParas` : *(int * int) → int list*
: Diese Funktion liefert die Menge aller Zeilennummern im angegebenen Intervall, auf welchen ein Paragraphenname definiert wird.

`allLoops` : *(int * int) → int list*
: Berechnet innerhalb des angegebenen Intervalls die Menge von Zeilennummern, auf welchen mindestens einmal das Schlüsselwort UNTIL resp. TIMES vorkommt.

`allSects`, `allDivs`, `allIfs`, `allEls`, `allGos` `allPerfs`, `allThrus` : *(int * int) → int list*
: Diese Funktionen liefern je eine Menge von Zeilennummern innerhalb eines Intervalls, auf welchen mindestens einmal ein bestimmtes Schlüsselwort vorkommt (SECTION, DIVISION, IF, ELSE, GO, PERFORM, THRU resp. THROUGH).

4 Erster Lösungsansatz

4.1 Program-Slicing anwenden auf den gesamten Programmtext

4.1.1 Kurzbeschreibung

1. *Kontrollfluss des Programms:* Da der Kontrollfluss eines Programms nicht aus der zeilen-orientierten Darstellung der Cross-Referencer-Daten alleine bestimmt werden kann, müssen alle Sprunganweisungen und Subroutinenaufrufe in das Program-Slice integriert werden. Nur so ist gewährleistet, dass die in das Slice übernommenen Anweisungen auch in der identischen Reihenfolge wie im Originalprogramm durchlaufen werden.

2. *Transitive Hülle bilden:* Es werden alle Zeilen des Programmtexts in das Program-Slice integriert, in welchem Variablen verwendet werden, die direkt oder indirekt auf mindestens eine der Variablen auf der Zeile des Slicing-Kriteriums Einfluss nehmen können.

3. *Erste syntaktische Ergänzung:* Alle bedingten Anweisungen, welche mindestens eine Anweisung aus dem Program-Slice enthalten, werden bestimmt und in das Program-Slice aufgenommen.

4. *Zweite syntaktische Ergänzung:* Als nächsten Schritt werden noch diejenigen Strukturen hinzugefügt, welche für die Sicherstellung der syntaktischen Korrektheit des Program-Slices erforderlich sind (z.B. Paragraphname und Paragraph-Endmarke). Konkret: alle Strukturen, welche Teile aus dem berechneten Program-Slice enthalten, werden ebenfalls aufgenommen.

4.1.2 Beschreibung der Funktionen

Die Hauptfunktion für diesen Ansatz bildet die Funktion **ProgramSlice1**, wie sie in Abbildung 5 dargestellt ist. Als Parameter übernimmt sie das Slicing-Kriterium. Da das Slicing-Kriterium immer in das zu generierende Program-Slice übernommen wird, kann sie auch als initiales Slice betrachtet werden.

ProgramSlice1 (slc)

BEGIN

$$
\begin{array}{ll}
gpP & := \texttt{allPairGos(MIN, MAX)} \cup \texttt{allPairPerfs(MIN, MAX)} \\
jps & := \bigcup_{p \in gpP} \texttt{fill}(p) \\
(slc, dots) & := \mathsf{transitiveClosureIfs}(\texttt{MIN}, \texttt{MAX}, (slc \cup jps), \emptyset) \\
Ps & := jps \cap \texttt{allProcs(MIN, MAX)} \\
Pn & := \bigcup_{l \in Ps} \texttt{getMset}(''\mathrm{ProcedureName}'', l) \\
Pl & := \bigcup_{p \in Pn} \texttt{getLset}(\texttt{MIN}, \texttt{MAX}, ''\mathrm{ParagraphName}'', p) \\
Stp & := (\texttt{paraPairs(MIN, MAX)} \cup \texttt{sectPairs(MIN, MAX)} \cup \texttt{divPairs(MIN, MAX)}) \\
Sts & := \{(x, y) \in Stp \mid \exists s \in (slc \cup Pl) : x \leq s \leq y\} \\
Stl & := \bigcup_{p \in Sts} \texttt{toSet}(p) \\
dots & := (dots \cup Stl) \setminus \\
 & \quad (slc \cup \texttt{allParas(MIN, MAX)} \cup \texttt{allSects(MIN, MAX)} \cup \texttt{allDivs(MIN, MAX)}) \\
slc & := (slc \cup Stl) \setminus dots
\end{array}
$$

RETURN $(slc, dots)$

END

Abbildung 5: **ProgramSlice1** (slc)

Bei einigen der Funktionen kann, wie wir noch sehen werden, ein relevanter Wirkungsbereich angegeben werden. Mit anderen Worten, wir verfügen über Funktionen, die ihre Informationen ausschliesslich in einem definierten Intervall suchen. Im vorliegenden Ansatz sind die Intervalle einstweilen so gewählt, dass die besagten Funktionen den gesamten Programmtext durchsuchen.

In einem ersten Schritt wird der gesamte Kontrollfluss des Programms als Gerüst in das Program-Slice aufgenommen. Dazu werden die beiden Funktionen `allPairGos` und `allPairPerfs` benutzt. Sie liefern je eine Menge von paarweisen Zeilennummern, welche angeben, wo Sprunganweisungen resp. Subroutinenaufrufe beginnen, und wo sie enden. Die Grössen `MIN` und `MAX` bezeichnen jeweils die Grenzen des Untersuchungsbereichs (vergl. Abschnitt 5.1). Nun besteht aber auch die Möglichkeit, dass sich Anweisungen über mehr als zwei Zeilen erstrecken; in diesem Falle müssen beim Integrieren in das Gerüst des Program-Slices nicht nur die Grenzen der Sprunganweisungen resp. Subroutinenaufrufe, sondern auch die Zeilennummern innerhalb der Intervallgrenzen übernommen werden.

Anschliessend kann das Slicing mit Hilfe der Funktion transitiveClosureIfs (siehe Abbildung 6) durchgeführt werden. Ihr Wirkungsintervall wird auf das ganze Programm festgesetzt. Die beiden letzten Parameter dieser Funktion repräsentieren das vorläufige Slice. Der eine beinhaltet die Zeilennummern, welche vollständig im aktuellen Slice liegen (Vereinigung des initialen Slices mit dem Gerüst) und der andere die Endmarken (noch leere Menge). Als Resultat erhält man ein neues slc und die neuen Endmarken ($dots$).

Um ein syntaktisch korrektes Programm zu erhalten, muss in einem weiteren Schritt das so entstandene Slice syntaktisch ergänzt werden. Erstens muss sichergestellt werden, dass jede in Sprunganweisungen resp. Subroutinenaufrufen verwendete Marke auch im Program-Slice vorkommt, und zweitens, dass jede Anweisung im gleichen Paragraphen erscheint wie im Originalprogramm. Ersteres wird erreicht, indem mit Hilfe der Funktion `allProcs` in allen zu einem Sprungbefehl gehörenden Zeilen (jps) die verwendeten Paragraphen bestimmt werden, und anschliessend diejenigen Zeilen in das Slice integriert werden, die den Anfang dieses Paragraphen markieren (Pl). Letzteres wird erreicht, indem für alle Paragraphen, Sections und Divisions getestet wird, ob mindestens eine Anweisung aus der entsprechenden Struktur im Program-Slice verwendet wird. Falls dem so ist, wird sie in das Slice übernommen.

Beim Integrieren von Endmarken (z.B von Paragraphen, Sections, ...) ist zu unterscheiden zwischen Zeilen, welche als ganzes im Slice Verwendung finden, und zwischen Zeilen, welche nur als Endmarke einer Struktur in das Slice übernommen werden. Diese Unterscheidung ist insofern relevant, als auf den Zeilen mit Endmarken ebenfalls Anweisungen stehen können, die aber für das Slice nicht benötigt werden und deshalb bei der Ausgabe der Endmarkenzeile nicht angezeigt werden sollen.

transitiveClosureIfs $(f, t, slc, dots)$
BEGIN
  REPEAT
    $slc' := \mathsf{transitiveClosureVar}(f, t, slc)$
    $ifs := \{(x, y, z) \in \texttt{allTripleIfs}(f, t) \mid \exists s \in slc' : x \leq s \leq z\}$
    $dots := (dots \cup \{z \mid (x, y, z) \in ifs\}) \setminus (slc' \cup \texttt{allEls}(f, t))$
    $slc := (slc' \cup \{x \mid (x, y, z) \in ifs\} \cup \{y \mid (x, y, z) \in ifs\}) \setminus dots$
  UNTIL $slc = slc'$
  RETURN $(slc, dots)$
END

Abbildung 6: **transitiveClosureIfs** $(f, t, slc, dots)$

Die Funktion **transitiveClosureIfs** in Abbildung 6 bildet als erstes die transitive Hülle der Abhängigkeiten aller Variablen, welche im Slice vorkommen (slc'), und zwar mit Hilfe der Funktion transitiveClosureVar (siehe Abbildung 7). Anschliessend werden sämtliche bedingten Anweisungen bestimmt, welche mindestens eine der Zeilen aus dem Slice in einem ihrer Äste enthalten. Sie werden durch eine Menge von Tripeln dargestellt (ifs) und müssen in das Slice integriert werden.

Durch das Integrieren von bedingten Anweisungen kann es vorkommen, dass mit den Bedingungen neue, im Slice noch nicht vorhandene Variablen auftreten. Ihre transitiven Abhängigkeiten müssen selbstverständlich ebenfalls in das Slice übernommen werden. Darum werden die oben genannten Schritte solange wiederholt, bis keine zusätzlichen Variablen aus Bedingungen mehr in das Slice übernommen werden.

transitiveClosureVar (f, t, ls)
BEGIN
    $vls := ls$
    WHILE $ls \neq \emptyset$ DO
        $ls' := \texttt{getLines}(f, t, \texttt{getMembers}(ls))$
        $ls := ls' \setminus vls$
        $vls := vls \cup ls'$
    END
    RETURN vls
END

Abbildung 7: **transitiveClosureVar** (f, t, ls)

In Abbildung 7 nun ist die Funktion **transitiveClosureVar** dargestellt. Sie ist für die Bildung der transitiven Hülle der Abhängigkeiten aller Variablen verantwortlich, welche auf mindestens einer der Zeilen in der Menge der Zeilennummern (ls) auftreten.

Die Funktion `getMembers` liefert die Menge der Variablennamen, welche innerhalb einer Menge von Zeilennummern vorkommen. Diese Menge von Variablennamen verwendet die Funktion `getLines`, um wiederum eine Menge von Zeilennummern zu bestimmen (ls'), auf welchen mindestens eine der Variablen vorkommt. Die so gefundenen Zeilennummern werden in der Menge vls kumuliert, wobei der Vorgang solange wiederholt wird, bis keine neuen Zeilennummern mehr gefunden werden, die nicht bereits in der Menge vls enthalten sind (d.h. die transitive Hülle bezüglich der Abhängigkeiten von ls ist komplett).

4.2 Beschränken des Program-Slicing auf einen Paragraphen

4.2.1 Kurzbeschreibung

Der im Abschnitt 4.1 beschriebene Ansatz beinhaltet den gesamten Kontrollfluss des Programms, was den Slice in aller Regel voluminös macht. Eine Möglichkeit, dies zu vermeiden, besteht darin, das Slicing auf einen begrenzten Bereich anzuwenden (z.B. Paragraphen). Es ist aber auch denkbar, das Slicing auf eine bedingte Anweisung oder sogar auf einen beliebigen Bereich innerhalb einer Sequenz von Anweisungen zu reduzieren. Wir beschränken zwar das Slicing auf einen Teilbereich des Programms, lassen aber dennoch die Einflüsse, ausgehend von aufgerufenen Paragraphen aus dem zu slicenden Programmteil, nicht unberücksichtigt. Nachfolgend zeigen wir das Vorgehen beim Slicing eines eingeschränkten Bereiches, d.h. eines Bereichs der kleiner ist als das gesamte Programm:

1. *Transitive Hülle bilden:* Entspricht dem gleichnamigen Punkt in Abschnitt 4.1.1, ausser dass der Untersuchungsbereich nicht das gesamte Programm umfasst, sondern einen enger definierten Bereich (z.B. einen Paragraphen).

2. *Erste syntaktische Ergänzung:* Als nächstes werden alle bedingten Anweisungen, welche mindestens eine Zeile aus der transitiven Hülle umschliessen, ebenfalls in das Slice übernommen.

3. *Kontrollfluss des Paragraphen:* Anschliessend werden alle Sprunganweisungen und Subroutinenaufrufe innerhalb des fraglichen Paragraphen in das Slice integriert, um gegebenenfalls auch Einflüsse, ausgehend von aufgerufenen Paragraphen, auf die Variablen in der Zeile des Slicing-Kriteriums berücksichtigen zu können.

4. *Zusätzliche Paragraphen:* Im folgenden Schritt werden alle Paragraphen, welche durch die Sprunganweisungen benutzt werden, bestimmt und einer zusätzlichen Untersuchung unterzogen. Dabei ist zu beachten, dass in allen Subroutinenaufrufen der Form PERFORM ... THRU die vollständige Paragraphen-Sequenz in die Menge der zu untersuchenden Paragraphen einfliesst.

5. *Repetieren:* Die vorangegangenen Schritte werden so oft wiederholt, bis sich keine zusätzlichen Paragraphen mehr finden, die für das Slicing geprüft werden müssen.

6. *Zweite syntaktische Ergänzung:* Schliesslich sind noch die SECTIONs und DIVISIONs zu bestimmen, in denen Zeilen auftreten, die im Program-Slice verwendet werden. Auch sie werden in das Slice übernommen.

4.2.2 Beschreibung der Funktionen

Im Gegensatz zum ursprünglichen Ansatz wird der Kontrollfluss hier nicht bereits beim Starten in das Slicing miteinbezogen, sondern wird separat für jeden Paragraphen betrachtet. Auch die implizite Übergabe der Kontrolle der Paragraphen untereinander durch das sogenannte *Fall-Through* bleibt unberücksichtigt, ausser sie wird durch ein explizites PERFORM ... THRU, welches im Slice vorkommt, vorgegeben. Dadurch werden grosse Teile des Kontrollflusses erst gar nicht in das Program-Slice aufgenommen, weshalb in der Regel die so generierten Slices einen wesentlich kleineren Umfang aufweisen als im früheren Ansatz.

Aus Gründen der besseren Beschreibbarkeit werden wir die Funktionen, die den im vorangegangenen Abschnitt beschriebenen Schritten entsprechen, in umgekehrter Reihenfolge vorstellen, da so die Vorgehensweise besser sichtbar gemacht werden kann.

ProgramSlice2 (slc)
BEGIN
REPEAT
$slc' := slc$
$ps := \{(x,y) \in \texttt{paraPairs(MIN,MAX)} \mid \exists s \in slc : x \leq s \leq y\}$
$(slc, dots) := \mathsf{transitiveClosureStructs}(ps, \emptyset, slc, \emptyset)$
$sts := \{(x,y) \in (\texttt{sectPairs(MIN,MAX)} \cup \texttt{divPairs(MIN,MAX)}) \mid \exists s \in slc : x \leq s \leq y\}$
$dots := (dots \cup \{y \mid (x,y) \in sts\}) \setminus slc$
$slc := (slc \cup \{x \mid (x,y) \in sts\}) \setminus dots$
UNTIL $slc = slc'$
RETURN $(slc, dots)$
END

Abbildung 8: **ProgramSlice2** (slc)

Abbildung 8 stellt die Hauptfunktion dieses Ansatzes für das Slicing dar. Die Funktion **ProgramSlice2** nimmt eine Menge von Programmzeilen entgegen. Beim Aufruf der Funktion besteht diese Menge aus genau einem Element, nämlich dem Slicing-Kriterium, zu welchen das Program-Slice gebildet werden soll.

Für die Zeilennummer in slc wird bestimmt, in welchem Paragraphen sie enthalten ist. Der betroffene Paragraph wird durch ein Paar von Zeilennummern repräsentiert. Der zu analysierende Paragraph und die interessierende Zeile werden der Funktion transitiveClosureStructs (siehe Abbildung 9) als Parameter übergeben. Diese Funktion berechnet das Program-Slice soweit, dass anschliessend nur noch die verwendeten Sections und Divisions syntaktisch zu ergänzen sind.

Nach bekanntem Muster werden schliesslich die beiden Mengen slc und $dots$ zur Repräsentation des Program-Slices berechnet.

Die verwendete Funktion $\mathsf{transitiveClosureStructs}$ analysiert jeden ihr übergebenen Paragraphen einzeln. Durch diese separate Bearbeitung bleiben aber mögliche Abhängigkeiten von Variablen aus unterschiedlichen Paragraphen unberücksichtigt, weshalb es unerlässlich ist, die in der Funktion $\mathsf{ProgramSlice2}$ beschriebenen Schritte solange zu wiederholen, bis sich bei einem erneuten Analysedurchlauf am Program-Slice keine Änderungen mehr zeigen.

$$
\begin{array}{l}
\textbf{transitiveClosureStructs}\ (pairs, analysedP, slc, dots)\\
\quad \text{BEGIN}\\
\quad\quad \text{WHILE } pairs \neq \emptyset \text{ DO}\\
\quad\quad\quad (f,t) \ := \texttt{max}(pairs)\\
\quad\quad\quad analysedP \ := analysedP \cup \{\texttt{max}(pairs)\}\\
\quad\quad\quad (slc, dots) \ := \mathsf{transitiveClosureGosPerfs}(f, t, slc, dots)\\
\quad\quad\quad uP \ := (slc \cap \texttt{allProcs}(f,t))\\
\quad\quad\quad uPn \ := \bigcup_{l \in uP} \texttt{getMset}(''\text{ProcedureName}'', l)\\
\quad\quad\quad uPl \ := \bigcup_{p \in uPn} \texttt{getLset}(\texttt{MIN}, \texttt{MAX}, ''\text{ParagraphName}'', p)\\
\quad\quad\quad uPp \ := \{(x,y) \in \texttt{paraPairs}(f,t) \mid \exists a \in uPl : a = x\}\\
\quad\quad\quad dots \ := (dots \cup \{t\} \cup \{y \mid (x,y) \in uPp)\}) \setminus (slc \cup \{x \mid (x,y) \in uPp)\}\\
\quad\quad\quad slc \ := (slc \cup \{f\} \cup \{t\} \cup \{x \mid (x,y) \in uPp)\}) \setminus dots\\
\quad\quad\quad thP \ := (slc \cap \texttt{allThrus}(f,t))\\
\quad\quad\quad thPn \ := \bigcup_{l \in thP} \{\texttt{getMset}(''\text{ProcedureName}'', l)\}\\
\quad\quad\quad thPl \ := \bigcup_{ps \in thPn} \{\bigcup_{p \in ps} \texttt{getLset}(\texttt{MIN}, \texttt{MAX}, ''\text{ParagraphName}'', p)\}\\
\quad\quad\quad thPp \ := \{(x,y) \in \texttt{paraPairs}(\texttt{MIN}, \texttt{MAX}) \mid \exists p \in thPl : \texttt{min}(p) \leq x \wedge y \leq \texttt{max}(p)\}\\
\quad\quad\quad pairs \ := (pairs \cup thPp \cup uPp) \setminus analysedP\\
\quad\quad \text{END}\\
\quad\quad \text{RETURN } (slc, dots)\\
\quad \text{END}
\end{array}
$$

Abbildung 9: **transitiveClosureStructs** $(pairs, analysedP, slc, dots)$

Die Funktion **transitiveClosureStructs**, wie sie in Abbildung 9 dargestellt ist, lässt das Program-Slice innerhalb der ihr übergebenen Intervalle (hier Paragraphen) bestimmen und ergänzt es um alle Zeilennummern derjenigen Sprungmarken (Paragraphnamen), welche im Slice verwendet werden.

Alle noch nicht untersuchten Paragraphen stehen in der Variablen $pairs$, diejenigen aber, welche bereits bearbeitet sind, werden in der Variablen $analysedP$ notiert. Die nachfolgenden Berechnungsschritte gelten immer für genau einen Paragraphen und werden darum so oft wiederholt, bis alle in $pairs$ vorhandenen Paragraphen analysiert sind.

Als erstes wird aus der Menge $pairs$ ein Paragraph bestimmt, der untersucht werden soll. Dabei spielt es keine Rolle, in welcher Reihenfolge die Paragraphen bearbeitet werden. Die Grössen f und t bezeichnen den Anfang resp. das Ende des ausgewählten Paragraphen.

Der selektierte Paragraph wird in $analysedP$ notiert und so vor wiederholtem Aufgreifen zur Analyse bewahrt.

Mit Hilfe der Funktion transitiveClosureGosPerfs (siehe Abbildung 10) werden jene Anweisungen des untersuchten Paragraphen in das Program-Slice übernommen, welche den Kontrollfluss beeinflussen können. Zur Sicherstellung der Korrektheit muss das zurückgelieferte Slice an dieser Stelle aber noch syntaktisch ergänzt werden. Da Sprunganweisungen und die Subroutinenaufrufe zusätzliche Paragraphen verwenden, müssen diese gesondert bestimmt (uPp) und ebenfalls in die Darstellung des Program-Slices (slc, $dots$) integriert werden.

Ergänzend ist noch zu beachten, dass ein Subroutinenaufruf grundsätzlich auch als eine Sequenz von Paragraphen auftreten kann, wobei selbstverständlich auch die Paragraphen innerhalb einer solchen Sequenz analysiert werden müssen. Eine solche Paragraph-Sequenz wird im Programmtext mit Hilfe des Schlüsselwortes THRU resp. THROUGH angegeben, wobei der Anfang und das Ende der Sequenz zusammen mit den entsprechenden Schlüsselworten jeweils auf einer gemeinsamen Zeile notiert sind. Die betroffenen Zeilennummern, in welchen solche Sequenzen auftreten, werden durch die Funktion `allThrus` geliefert und bilden die Grundlage zur Bestimmung der fraglichen Paragraphen ($thPp$).

Die Paragraphen aus uPp und $thPp$ werden anschliessend in die Menge der noch zu untersuchenden Paragraphen aufgenommen, wobei aber sichergestellt wird, dass bereits analysierte Paragraphen ($analysedP$) nicht erneut bearbeitet werden.

transitiveClosureGosPerfs $(f, t, slc, dots)$
BEGIN
$gp \quad := \texttt{allPairGos}(f,t) \cup \texttt{allPairPerfs}(f,t)$
$dots' \quad := dots \setminus \{y \mid (x,y) \in gp\}$
$slc' \quad := slc \cup (\bigcup_{p \in gp} \texttt{fill}(p))$
$(slc, dots) := \mathsf{transitiveClosureIfs}(f, t, slc', dots')$
RETURN $(slc, dots)$
END

Abbildung 10: **transitiveClosureGosPerfs** $(f, t, slc, dots)$

Mit der Funktion **transitiveClosureGosPerfs** in der Abbildung 10 werden alle Subroutinenaufrufe und Sprunganweisungen innerhalb des Untersuchungsbereichs mit den Grenzen f und t bestimmt (entspricht hier einem Paragraphen) und in das Slice (slc', $dots'$) integriert. Dieser wird als Parameter an die Funktion transitiveClosureIfs weitergegeben, wo das Program-Slice weiter ergänzt wird.

Die ersten beiden Schritte aus der Kurzbeschreibung zu diesem Ansatz (siehe Abschnitt 4.2.1), nämlich *das Bilden der transitiven Hülle* und *die erste syntaktische Ergänzung*, werden mit den gleichen Funktionen transitiveClosureVar und transitiveClosureIfs bearbeitet, wie sie bereits im vorangehenden Ansatz beschrieben wurden (siehe Abbildungen 6 und 7).

5 Verbesserter Lösungsansatz

5.1 Begriffe und Vorgehen

Der in Abschnitt 4.2 vorgestellte Ansatz, mit dem auf einen Paragraphen begrenzten Program-Slicing, erzeugt bereits viel genauere Slices als sein Vorgänger. Im folgenden wollen wir nun einen Schritt weitergehen und zeigen, wie sich der Umfang der Slices noch weiter reduzieren lässt.

Bevor wir dies im einzelnen ausführen, sollen aber noch zwei Begriffe erläutert werden, mit deren Hilfe wir im weiteren die Verbesserung begründen, nämlich der des *Untersuchungsbereichs* und der des *Betrachtungsbereichs*.

Der *Untersuchungsbereich* wird im Zusammenhang mit den Funktionen verwendet. Er ist eine technische Grösse und gibt an, in welchem Intervall eine Funktion ihre Analyse durchführen soll. Dem gegenüber bestimmt der *Betrachtungsbereich*, auf welchem Teilstück des Programms das eigentliche Slicing operieren soll. Zur Veranschaulichung soll das Beispiel in Abbildung 11 dienen, wobei wir vorausschicken, dass die Zeile 9 die interessierenden Variablen enthält.

```
              PROGRAM prg            PROGRAM prg            PROGRAM prg
 1            PARA1                  PARA1                  PARA1
 2                MOVE m TO n
 3                MOVE a TO b            MOVE a TO b            MOVE a TO b
 4            PARA2                  PARA2
 5                MOVE z TO b            MOVE z TO b
 6                MOVE w TO b            MOVE w TO b
 7            PARA3                  PARA3                  PARA3
 8                MOVE w TO x            MOVE w TO x
 9 -->            MOVE b TO a            MOVE b TO a            MOVE b TO a
10                MOVE z TO y            MOVE z TO y
11                PERFORM PARA1          PERFORM PARA1          PERFORM PARA1

                      (a)                    (b)                    (c)
```

Abbildung 11: Beispiel zu Betrachtungs- und Untersuchungsbereich

Das Originalprogramm, welches analysiert werden soll, ist in der Kolonne `(a)` dargestellt. Die Programme `(b)` und `(c)` zeigen die entsprechenden Program-Slices, wie sie aufgrund des ersten resp. des zweiten Ansatzes berechnet werden (siehe Abschnitte 4.1 und 4.2).

- Im ersten Ansatz gilt als Betrachtungsbereich der gesamte Programmtext, da wir das Slice für das ganze Programm ermitteln wollen. Auch der Untersuchungsbereich entspricht dem gesamten Programm, weil die Funktion transitiveClosureVar die transitive Hülle innerhalb des *ganzen* Programmtexts berechnet.

  Mit anderen Worten, die Argumente f und t der Funktion transitiveClosureVar (vergl. Abbildung 7), welche den Untersuchungsbereich festlegen, sind auf die erste resp. letzte Zeile des Programms gesetzt. Dies bewirkt, dass im ersten Schritt die Zeilen `5` und `6` in die transitive Hülle übernommen werden und im folgenden Schritt dann auch die Zeilen `8` und `10`. Diese zuletzt integrierten Zeilen haben aber auf die Variablen in der interessierenden Zeile `9` weder einen direkten noch einen indirekten Einfluss. Das resultierende Prorgam-Slice ist in Abbildung 11.`b` dargestellt.

- Im zweiten Ansatz interessieren wir uns ausschliesslich für die Einflüsse auf das Slicing-Kriterium innerhalb des fraglichen Paragraphen, weshalb in diesem Fall sowohl der Betrachtungsbereich als auch der Untersuchungsbereich auf die Grösse eines Paragraphen festgesetzt wurden. Dadurch haben wir erreicht, dass die Funktion transitiveClosureVar die transitive Hülle nur innerhalb eines Paragraphen zu berechnen hat, was in der Regel beträchtliche Auswirkungen auf das Volumen der berechneten Slices hat. Gemäss dem zweiten Ansatz muss nämlich zuerst der Paragraph analysiert werden, in welchem das Slicing-Kriterium liegt (`PARA3`), und anschliessend die durch Sprunganweisungen oder Subroutinenaufrufe benutzten Paragraphen (`PARA1`). Der Paragraph `PARA3` vergrössert die transitive Hülle nicht, da keine andere Zeile innerhalb dieses Paragraphen die Variable `a` resp. `b` enthält, ausser das Slicing-Kriterium selbst. Die Analyse des Paragraphen `PARA1` zeigt, dass einzig die Zeile `3` zusätzlich in den Slice aufgenommen werden muss. Das endgültige Program-Slice findet sich in Abbildung 11.`c`.

Diese Erklärungen machen deutlich, dass die Verringerung des Umfangs der Program-Slices im zweiten Ansatz gegenüber denjenigen im ersten darauf beruht, dass der Untersuchungsbereich für die Bildung der transitive Hülle der Abhängigkeiten stark reduziert wurde.

Es stellt sich nun die Frage, ob es möglich ist, das Untersuchungsintervall noch weiter einzuschränken, während der Betrachtungsbereich unverändert auf einen Paragraphen fokussiert bleibt.

Bis anhin musste die Reihenfolge von Anweisungen nicht in die Betrachtungen einbezogen werden. Wir haben den Untersuchungsbereich und den Betrachtungsbereich bisher bewusst auf das selbe Intervall beschränkt, weil in dieser Konstellation die Reihenfolge der Anweisungen keine Rolle spielt und bei der Analyse vernachlässigt werden kann. Falls nun der Untersuchungsbereich aber kleiner gewählt wird als der Betrachtungsbereich, stellt sich das Problem, wie die Untersuchungsbereiche innerhalb des grösseren Betrachtungsbereichs kombiniert werden müssen, damit der Betrachtungsbereich vollständig analysiert wird.

Im Wesentlichen sind hierbei folgende zwei Aspekte zu berücksichtigen. Es muss nämlich definiert werden,

- wie der Betrachtungsbereich in mehrere Untersuchungsbereiche *aufzuspalten* ist.
- in welcher *Reihenfolge* die Untersuchungsbereiche analysiert und in das Program-Slicing einbezogen werden.

5.1.1 Aufspalten des Betrachtungsbereichs

Ziel beim Aufteilen des Betrachtungsbereichs in kleinere Untersuchungsbereiche ist, den Zuwachs des Program-Slices durch die Bildung der transitiven Hülle, welche für einen Untersuchungsbereich als Ganzes berechnet wird, zu minimieren. Aus diesem Grund müssen letztere möglichst klein gewählt werden. Als Hilfe für die Aufgliederung bietet sich der Kontrollfluss an. Jede Stelle im Programm, wo sich der Kontrollfluss verzweigt resp. vereint, ist ein markanter Punkt, wo das Program-Slicing besondere Opertionen durchführen muss. In diesem Zusammenhang verweisen wir auf [ASU88], worin ein *Grundblock* in einem Programmtext definiert wird als eine Folge von fortlaufenden Anweisungen, in die der Kontrollfluss am Anfang eintritt, und die er am Ende verlässt, ohne dass er dazwischen anhält oder – ausser am Ende – verzweigt.

Wir werden für eine unserer Einheiten, nämlich das ATOM, eine ähnliche Definition verwenden, mit dem Unterschied, dass bei einem ATOM (im Gegensatz zum *Grundblock* in [ASU88]) keine Verzweigung beim Verlassen zugelassen ist.

Weitere Einheiten (siehe Abbildung 12) enthalten COBOL-spezifische Sprachkonstrukte, welche den Kontrollfluss beeinflussen können. Es sind dies die Einheiten GOTOSTMT, PERFORMSTMT und IFSTMT. Die ersten beiden stellen einzelne Anweisungen dar, die

Part	=	Atom | Block | Chunk
Chunk	=	[Atom | Block] IfStmt
		{[Atom | Block] IfStmt} [Atom | Block]
Block	=	[Atom] (PerformStmt | GotoStmt)
		{[Atom] (PerformStmt | GotoStmt)} [Atom]
IfStmt	=	(*expression*, Part, Part)
Atom	=	{SimpleStmt}
GotoStmt	=	*goto-statement*
PerformStmt	=	*perform-statement*
SimpleStmt	=	*simple-statements* (z.B. move-Anweisung)

Abbildung 12: Aufteilen eines Betrachtungsbereichs in Untersuchungsbereiche

bei der Analyse speziell zu behandeln sind. Die dritte Einheit ist ein Tripel und stellt eine bedingte Anweisung dar.

Mit Hilfe der vorgenannten einfachen Einheiten werden weitere definiert. So ist ein Block z.B. eine Einheit, welche mindestens ein GotoStmt oder ein PerformStmt, aber kein IfStmt enhält. Ein Chunk hingegen ist eine Einheit, die mindestens ein IfStmt enhält. Die Einheit mit der grössten Generalität heisst Part. Sie kann sowohl ein Atom, als auch ein Block oder sogar ein Chunk sein. Die Beziehungen zwischen den Einheiten lassen sich, wie in Abbildung 12 festgehalten, darstellen.

Der Betrachtungsbereich in unserem verbesserten Ansatz ist identisch mit demjenigen des zweiten Ansatzes, d.h. er umfasst genau einen Paragraphen, und wird für die Analyse in der gleichen Art und Weise wie die Einheit Part aufgespalten.

5.1.2 Sequenz der Untersuchungsbereiche für rückwärts-orientierte Slices

Neu ist an diesem verbesserten dritten Ansatz, dass erstmals die Richtung des Program-Slicing mit eine Rolle spielt. Die Reihenfolge, in der die kleineren Untersuchungsbereiche bearbeitet werden sollen, hängt vom gewünschten Resultat des Program-Slicings ab. Da wir als Ziel verfolgen, rückwärts-orientierte Slices zu erhalten, welche die Effekte auf eine bestimmte Programmzeile erfassen, wird in unserem Ansatz die Analyse ebenfalls rückwärts, d.h. von hinten nach vorne arbeiten. Dabei nutzen wir die folgenden Informationen zu den einzelnen Klassen von Anweisungen:

- *Einfache Anweisungen:* Eine Sequenz von einfachen Anweisungen wird beim Ausführen des Programms in der Reihenfolge durchlaufen, wie die einzelnen Anweisungen im Programmtext aufgeführt sind. Da gemäss unseren Voraussetzungen jede einfache Anweisung auf einer separaten Zeile steht, lässt sich ihr eine eindeutige

Zeilennummer zuordnen. Die aufsteigend sortierten Zeilennummern der einfachen Anweisungen entsprechen demnach der Reihenfolge während des Programmablaufs.

Bei der Berechnung eines rückwärts-orientierten Program-Slices werden sie deshalb in umgekehrter Reihenfolge (absteigende Zeilennummern) analysiert.

- *Bedingte Anweisung:* Beim Ausführen einer bedingten Anweisung wird zuerst die Bedingung ausgewertet und anschliessend entweder die eine oder die andere Sequenz von Anweisungen ausgeführt.

  Entscheidend für das Program-Slicing ist hier, dass die Anweisungssequenzen der beiden Äste voneinander unabhängig analysiert werden, und erst nachher die Bedingung in die Analyse miteinbezogen wird.

- *Sprunganweisung:* Gemeint ist hier ein unbedingter Sprung (z.B. GO TO in COBOL), der den Kontrollfluss an diejenige Stelle im Programm lenkt, welche durch eine Sprungmarke (hier ein Paragraphenname) bezeichnet ist. Alle auf eine Sprunganweisung folgenden Anweisungen bis zum Ende der aktiven Umgebung (scope) sind für den Kontrollfluss unerreichbar (toter Code).

  Für das Program-Slicing bedeutet dies, dass die Analyse an der Stelle im Programm fortgeführt wird, wo sich die Sprungmarke befindet. Die Anweisungen nach einer Sprunganweisung können vernachlässigt werden, da sie niemals einen Beitrag zum Slice beisteuern.

- *Subroutinenaufruf:* Hier wird der Kontrollfluss zwar an eine andere Stelle innerhalb des Programms gelenkt, kehrt aber, nachdem die Bearbeitung des Paragraphen abgeschlossen ist, wieder an den Ausgangspunkt zurück.

  Die unserem Projekt zugrundeliegende Zielsprache ist COBOL, welche bekanntlich einige Besonderheiten aufweist. So können in COBOL beispielsweise nur globale Variablen verwendet werden und die Paragraphen entsprechen den parameterlosen Prozeduren in anderen Programmiersprachen. Des weiteren kennt COBOL keine Schleifen. Sie müssen mit Hilfe von Subroutinenaufrufen (PERFORMs) gemeinsam mit den beiden Schlüsselwörtern UNTIL resp. TIMES implementiert werden. Eine weitere Besonderheit ist, dass Paragraphen in verschiedenen, sich überlappenden Paragraphen-Sequenzen verwendet werden können.

  Für das Program-Slicing hat dies zur Folge, dass bei jedem Subroutinenaufruf noch zusätzlich überprüft werden muss, ob er eine Schleife implementiert oder nicht.

5.2 Beschreibung der Funktionen

Die Hauptfunktion **ProgramSlice3** in Abbildung 13 dieses Ansatzes ist so gestaltet, dass sie die Program-Slices zu mehreren Slicing-Kriterien gleichzeitig berechnet und anschliessend miteinander vereint. Der Eingabeparameter ls ist deshalb eine Liste von Zeilennummern.

ProgramSlice3 (ls)

$$
\begin{array}{l}
\textsc{begin}\\
\quad ps \quad := \{(x,y) \in \texttt{paraPairs(MIN,MAX)} \mid \exists l \in ls : x \leq l \leq y\}\\
\quad (slc, dots) := (\emptyset, \emptyset)\\
\quad \textsc{while}\ ps \neq \emptyset\ \textsc{do}\\
\quad\quad (f,t) \ := \texttt{max}(ps)\\
\quad\quad ls' \quad := \{l \in ls \mid f \leq l \leq t\}\\
\quad\quad ifs \quad := \texttt{allTripleIfs}(f,t)\\
\quad\quad oIfs \ := \{(x,y,z) \in ifs \mid \nexists (u,v,w) \in (ifs \setminus \{(x,y,z)\}) : u \leq x \wedge z \leq w\}\\
\quad\quad scope \ := \{(x,z) \mid \exists l \in ls' : (x,y,z) \in oIfs \wedge x \leq l \leq z \ \wedge l = \texttt{max}(ls')\}\\
\quad\quad sct \quad := \begin{cases} \#2(\texttt{max}(scope)) & : \ scope \neq \emptyset \\ \texttt{max}(ls') & : \ sonst \end{cases}\\
\quad\quad (slc', dots') \ := \texttt{analyseParts}(f, sct, ls', \emptyset)\\
\quad\quad (slc, dots) \quad := \texttt{joinStates}((slc, dots), (slc', dots'))\\
\quad\quad ps \quad := ps \setminus \{(f,t)\}\\
\quad \textsc{end}\\
\quad structs \ := \{(x,y) \in (\texttt{divPairs(MIN,MAX)} \cup \texttt{sectPairs(MIN,MAX)}) \mid \exists s \in slc : x \leq s \leq y\}\\
\quad dots \ := (dots \cup \{y \mid (x,y) \in structs\}) \setminus slc\\
\quad slc \quad := (slc \cup \{x \mid (x,y) \in structs) \setminus dots\\
\quad \textsc{return}\ (slc, dots)\\
\textsc{end}
\end{array}
$$

Abbildung 13: **ProgramSlice3** (ls)

Zunächst müssen alljene Paragraphen bestimmt werden, welche mindestens eine der interessierenden Zeilennummern enthalten (ps).

Vor dem Analysieren aller Paragraphen wird das Program-Slice initialisiert. Die Reihenfolge der Bearbeitung dieser Paragraphen spielt keine Rolle. Im vorliegenden Fall kommt die absteigende Reihenfolge zum Zuge, aber wir könnten sie statt dessen in aufsteigender oder in einer zufälligen Reihenfolge analysieren, ohne dass damit das Resultat verändert würde, denn die Analysen der Paragraphen untereinander sind unabhängig. Jeder einzelne Paragraph wird wie folgt bearbeitet:

Zuerst werden alle Slicing-Kriterien bestimmt (ls'), welche innerhalb des zu analysierenden Paragraphen liegen. Es ist nicht erforderlich den gesamten Paragraphen zu analysieren, sondern nur den Bereich zwischen seinem Anfang und dem letzten Slicing-

Kriterium in ls'. Hierbei ist aber zu beachten, dass dieses auch innerhalb einer bedingten Anweisung liegen kann. In diesem Fall wird die Analyse bis zum Ende der bedingten Anweisung durchgeführt. Die Variable sct, welche die Nummer der letzten noch zu analysierenden Zeile enthält, wird entsprechend gesetzt.

Anschliessend wird das Slice $(slc', dots')$ für diesen Paragraphen durch die Funktion analyseParts (siehe Abbildung 14) berechnet. Als obere Grenze des Untersuchungsbereichs gilt der Beginn des Paragraphen und als untere Grenze die Zeile sct. Die Slicing-Kriterien ls' werden als initiales Slice betrachtet. Jedes so berechnete Slice wird dann in $(slc, dots)$ durch die Funktion `joinStates` kumuliert.

Nachdem nun alle Paragraphen aus ps analysiert sind, wird das Resultat syntaktisch ergänzt durch die Sections und Divisions, welche mindestens eine Zeile aus dem bis hierhin berechneten Slice enthalten.

analyseParts $(f, t, slc, dots)$
BEGIN

$$(slc, dots) := \begin{cases} \mathsf{analyseChunk}(f, t, slc, dots) & : \ \texttt{allIfs}(f,t) \neq \emptyset \\ \mathsf{analyseBlock}(f, t, slc, dots) & : \ \texttt{allIfs}(f,t) = \emptyset \wedge \\ & \quad (\texttt{allPerfs}(f,t) \neq \emptyset \vee \texttt{allGos}(f,t) \neq \emptyset) \\ \mathsf{analyseAtom}(f, t, slc, dots) & : \ \textit{sonst} \end{cases}$$

RETURN $(slc, dots)$
END

Abbildung 14: **analyseParts** $(f, t, slc, dots)$

Die Funktion **analyseParts** in Abbildung 14 lässt das Program-Slice durch eine der Funktionen analyseChunk, analyseBlock oder analyseAtom (siehe weiter unten) berechnen. Die Entscheidung, welche Funktion zur Anwendung kommt, hängt davon ab, ob bedingte Anweisungen, Subroutinenaufrufe oder Sprunganweisungen im Untersuchungsbereich vorkommen.

Sobald die Funktion **analyseChunk** in Abbildung 15 aufgerufen wird, steht zugleich fest, dass innerhalb des Untersuchungsbereichs (f, t) mindestens eine bedingte Anweisung existiert. Als erstes müssen alljene bedingten Anweisungen $oIfs$ bestimmt werden, welche nicht innerhalb von anderen bedingten Anweisungen liegen. Anschliessend wird mit Hilfe von $oIfs$ der Untersuchungsbereich durch die Funktion `makePairs` in weitere Teilbereiche aufgegliedert.

Das Program-Slicing analysiert diese Bereiche rückwärts, d.h. es wird immer das letzte Intervall als erstes analysiert. Bereiche, welche eine bedingte Anweisung enthalten müssen

analyseChunk $(f, t, slc, dots)$

$$
\begin{array}{l}
\textsc{begin} \\
\quad ifs \quad := \texttt{allTripleIfs}(f,t) \\
\quad oIfs \quad := \{(x,y,z) \in ifs \mid \nexists (u,v,w) \in (ifs \setminus \{(x,y,z)\}) : u \leq x \wedge z \leq w\} \\
\quad pairs \quad := \texttt{makePairs}(f, \{(x,z) \mid (x,y,z) \in oIfs\}, t) \\
\quad \textsc{while}\ pairs \neq \emptyset\ \textsc{do} \\
\quad\quad (v,w) \ := \texttt{max}(pairs) \\
\quad\quad pairs \ := pairs \setminus \{(v,w)\} \\
\quad\quad (slc, dots) := \begin{cases} \mathsf{analyseIfStmt}(v,w,slc,dots) & : \ \texttt{allIfs}(v,w) \neq \emptyset \\ \mathsf{analyseBlock}(v,w,slc,dots) & : \ \texttt{allIfs}(v,w) = \emptyset \wedge \\ & \quad (\texttt{allPerfs}(v,w) \neq \emptyset \vee \texttt{allGos}(v,w) \neq \emptyset) \\ \mathsf{analyseAtom}(v,w,slc,dots) & : \ sonst \end{cases} \\
\quad \textsc{end} \\
\quad \textsc{return}\ (slc, dots) \\
\textsc{end}
\end{array}
$$

Abbildung 15: **analyseChunk** $(f, t, slc, dots)$

durch die Funktion analyseIfStmt (siehe Abbildung 16) analysiert werden. Die übrigen werden, falls sie einen Subroutinenaufruf resp. eine Sprunganweisung enthalten, durch die Funktionen analyseBlock, ansonsten durch die Funktion analyseAtom analysiert.

analyseIfStmt $(f, t, slc, dots)$

$$
\begin{array}{l}
\textsc{begin} \\
\quad ifs \quad := \texttt{allTripleIfs}(f,t) \\
\quad (a,b,c) \quad := \texttt{min}\{(x,y,z) \in ifs \mid \nexists (u,v,w) \in (ifs \setminus \{(x,y,z)\}) : u \leq x \wedge z \leq w\} \\
\quad (slcT, dotsT) := \mathsf{analyseParts}(a+1, b, slc, dots) \\
\quad (slcF, dotsF) := \mathsf{analyseParts}(b+1, c, slc, dots) \\
\quad (slc', dots') \quad := \texttt{joinStates}((slcT, dotsT), (slcF, dotsF)) \\
\quad dots'' \ := (dots' \cup \{c\}) \setminus (slc' \cup (\{b\} \setminus \{c\})) \\
\quad slc'' \quad := (slc' \cup \{a\} \cup \{b\} \cup \{c\}) \setminus dots'' \\
\quad (slc, dots) \quad := \begin{cases} (slc, dots) & : \ slc' \cap \texttt{fill}(f,t) = \emptyset \\ (slc'', dots'') & : \ sonst \end{cases} \\
\quad \textsc{return}\ (slc, dots) \\
\textsc{end}
\end{array}
$$

Abbildung 16: **analyseIfStmt** $(f, t, slc, dots)$

Die Funktion **analyseIfStmt** in Abbildung 16 bestimmt als erstes diejenige bedingte Anweisung im Untersuchungsbereich, welche in keiner anderen bedingten Anweisung enthalten ist. Jede bedingte Anweisung wird als Tripel dargestellt. Dabei bezeichnet die erste Zeilennummer die Position des Schlüsselwortes IF, die zweite Zeilennummer die Position des Schlüsselwortes ELSE und die dritte Zeilennummer, wo die bedingte Anweisung endet.

Da wir das Programm nur statisch analysieren, müssen jeweils beide Äste betrachtet werden, weil nicht entschieden werden kann, welcher Ast beim Ausführen des Programms verwendet wird. Beide Äste werden unabhängig voneinander bearbeitet, und die erhaltenen Resultate durch die Funktion `joinStates` zum Slice $(slc', dots')$ kombiniert. Dieses Ergebnis wird um die Strukturinformationen der bedingten Anweisung (z.B. Anfangs- und Endzeile) ergänzt, welche durch das Slicing der beiden Äste nicht notwendigerweise in das Slice aufgenommen wurden. Das neue Slice wird aber nur dann als Resultat weitergeleitet, wenn mindestens eine Programmzeile aus einer der beiden untersuchten Äste für das Slicing relevant ist.

analyseBlock $(f, t, slc, dots)$
BEGIN
$(gf, gt) := \texttt{min}(\texttt{allPairGos}(f, t))$
$perfs := \texttt{allPairPerfs}(f, \texttt{min}(t, gf))$
$pairs := \texttt{makePairs}(f, perfs \cup \{(gf, gt)\}, \texttt{min}(t, gt))$
WHILE $pairs \neq \emptyset$ DO
$(v, w) := \texttt{max}(pairs)$

$$(slc, dots) := \begin{cases} \mathsf{analysePerform}(v, w, slc, dots) & : \ allPerfs(v, w) \neq \emptyset \\ \mathsf{analyseGoto}(v, w, slc, dots) & : \ allPerfs(v, w) = \emptyset \wedge \\ & \quad allGos(v, w) \neq \emptyset \\ \mathsf{analyseAtom}(v, w, slc, dots) & : \ sonst \end{cases}$$

$pairs := pairs \setminus \{(v, w)\}$
END
RETURN $(slc, dots)$
END

Abbildung 17: **analyseBlock** $(f, t, slc, dots)$

Die Funktion **analyseBlock** in Abbildung 17 spaltet den Untersuchungsbereich, ähnlich wie in der Funktion analyseChunk, in Teilbereiche. Anstelle von bedingten Anweisungen werden hier allerdings die Sprunganweisungen und die Subroutinenaufrufe als Teilungs-Kriterien verwendet. Die Analyse der Teilbereiche erfolgt auch hier wieder von hinten nach vorne, wobei das Resultat jeweils zum Vorgänger weitergereicht wird.

Die Tatsache, dass die Anweisungen nach einer Sprunganweisung nicht mehr erreichbar sind (toter Code), kann hier genutzt werden: Der Block muss nur bis zum Ende der ersten darin enthaltenen Sprunganweisung in Teilbereiche unterteilt werden; die nachfolgenden Anweisungen sind für die Analyse ohne Belang und bleiben unberührt.

Je nachdem, ob die Teilbereiche einen Subroutinenaufruf, eine Sprunganweisung oder nur einfache Anweisungen enthalten, werden sie durch die Funktionen analysePerform, analyseGoto oder analyseAtom analysiert.

analysePerform $(f, t, slc, dots)$

BEGIN

$$
\begin{array}{ll}
uPn & := \bigcup_{l \in \texttt{allProcs}(f,t)} \texttt{getMset}(''\text{ProcedureName}'', l) \\
uPl & := \bigcup_{pn \in uPn} \texttt{getLset}(\texttt{MIN}, \texttt{MAX}, ''\text{ParagraphName}'', pn) \\
uPp & := \{(x,y) \in paraPairs(\texttt{MIN}, \texttt{MAX}) \mid \texttt{min}(uPl) \leq x \leq \texttt{max}(uPl)\} \\
first & := \texttt{min}(uPp) \\
last & := \texttt{max}(uPp) \\
(v, \_) & := first \\
(\_, w) & := last \\
(slc', dots') & := \mathsf{analyseParagraphSeq}(uPp, slc, dots) \\
n & := \sum_{p \in uPp} \mid \texttt{fill}(p) \mid \\
(slc1', dots1') & := \begin{cases} (slc', dots') & : \ (\texttt{allLoops}(f,t) = \emptyset) \vee (slc' = slc) \\ \texttt{apply}(n, \mathsf{analyseParagraphSeq}, \\ \qquad uPp, slc', dots') & : \ sonst \end{cases} \\
dots'' & := (dots1' \cup (\{y \mid (x,y) \in \{first, last\}\} \setminus \{x \mid (x,y) \in \{first, last\}\})) \setminus slc1' \\
slc'' & := (slc1' \cup (\bigcup_{p \in \{first, last\}} \texttt{toSet}(p)) \cup \texttt{fill}(f,t)) \setminus dots'' \\
(slc, dots) & := \begin{cases} (slc, dots) & : \ (slc1' = slc) \wedge (slc \cap \texttt{fill}(f,t) = \emptyset) \\ (slc'', dots'') & : \ sonst \end{cases}
\end{array}
$$

RETURN $(slc, dots)$

END

Abbildung 18: **analysePerform** $(f, t, slc, dots)$

Die Funktion **analysePerform** in der Abbildung 18 ist für das Analysieren eines Subroutinenaufrufs zuständig. Wie die nachstehende Liste zeigt, ist ein Subroutinenaufruf in COBOL sehr variantenreich:

- Ein einzelner Paragraph wird als Unterprogramm verwendet.
- Eine Sequenz von aufeinander folgenden Paragraphen wird als Unterprogramm verwendet, wobei der erste und der letzte Paragraph der Sequenz jeweils links resp. rechts eines der Schlüsselwörter THRU resp. THROUGH steht.
- Ein einzelner Paragraph wird als Schleifenkörper verwendet, unter Benutzung einer der beiden Schlüsselwörter UNTIL resp. TIMES.
- Eine Sequenz von Paragraphen wird als Schleifenkörper verwendet.

Der erste Schritt besteht nun darin, die Sequenz uPp der aufeinander folgenden Paragraphen zu bestimmen, die durch den Subroutinenaufruf verwendet werden. Mitunter kann diese Paragraphensequenz, wie aus den oben genannten Varianten ersichtlich ist, genau aus einem Paragraphen bestehen.

Mit Hilfe der Funktion analyseParagraphSeq (siehe Abbildung 19) wird diese Sequenz analysiert und das Zwischenergebnis $(slc', dots')$ erzeugt.

Falls der Subroutinenaufruf verwendet wird, um eine Schleife zu implementieren, muss die Sequenz mehrmals analysiert werden. Der Grund dafür ist, dass bei Anweisungen innerhalb einer Schleife ihre Vorgänger auch ihre Nachfolger sind. So können Abhängigkeiten zwischen Variablen oftmals erst nach mehreren Durchläufen sichtbar werden. Um dem Rechnung zu tragen, muss also die Analyse des Schleifenkörpers (Sequenz von Paragraphen) so oft wiederholt werden, bis alle potentiellen Abhängigkeiten erfasst sind. Als obere Grenze für die Anzahl Wiederholungen der Analyse wird in [Hau89] die Anzahl der Anweisungen innerhalb der Schleife vorgeschlagen. Diese Grenze wird auch im vorliegenden Ansatz zur Anwendung gebracht.

Abschliessend wird noch geprüft, ob mindestens eine Zeile aus der Paragraphensequenz in den Program-Slice integriert wurde; wenn nicht, wird das ursprüngliche Slice unverändert zurückgegeben. Sind aber Teile aus den Paragraphen übernommen worden, muss sicher gestellt werden, dass auch diejenigen Paragraphen, welche im Subroutinenaufruf direkt angesprochen werden, im Slice vertreten sind, da andernfalls ein syntaktisch fehlerhafter Programmtext resultiert.

analyseParagraphSeq $(pairs, slc, dots)$
BEGIN
  WHILE $pairs \neq \emptyset$ DO
    $(f,t) \quad := \mathtt{max}(pairs)$
    $(slc', dots') \quad := \mathsf{analyseParts}(f, t, slc, dots)$
    $(slc, dots) := \begin{cases} (slc, dots) & : \quad (slc' \cap \mathtt{fill}(f,t) = \emptyset) \\ (slc' \cup \{f\}, dots') & : \quad sonst \end{cases}$
    $pairs \quad := pairs \setminus \{(f,t)\}$
  END
  RETURN $(slc, dots)$
END

Abbildung 19: **analyseParagraphSeq** $(pairs, slc, dots)$

Die Funktion **analyseParagraphSeq** in Abbildung 19 führt ein Slicing über den Bereich einer Sequenz von Paragraphen durch, wobei die Analyse wiederum rückwärts, von hinten nach vorne arbeitet. Leistet ein analysierter Paragraph einen Beitrag zum Program-Slice, der durch die Funktion analyseParts berechnet wurde, so wird auch jene Zeile, auf welcher der Paragraphennamen steht, mit in das Slice übernommen.

analyseGoto $(f, t, slc, dots)$
BEGIN
$gPn \ := \bigcup_{l \in \texttt{fill}(f,t)} \texttt{getMset}(''\text{ProcedureName}'', l)$
$gPl \ := \bigcup_{pn \in gPn} \texttt{getLset}(\texttt{MIN}, \texttt{MAX}, ''\text{ParagraphName}'', pn)$
$gPp \ := \{(x,y) \in \texttt{paraPairs}(\texttt{MIN}, \texttt{MAX}) \mid x \in gPl\}$
$(slc'', dots'') \ := (slc, dots)$
WHILE $gPp \neq \emptyset$ DO
$(v, w) \ := \texttt{max}(pairs)$
$(slc', dots') \ := \text{analyseParts}(v, w, slc, dots)$
$(slc'', dots'') \ := \texttt{joinStates}((slc'', dots''), (slc', dots'))$
$gPp \ := gPp \setminus \{(v, w)\}$
END
$(slc', dots') \ := (slc'' \cup \texttt{fill}(f,t), dots'')$
$dots \ := (dots' \cup (\{y \mid (x,y) \in gPp\} \setminus \{x \mid (x,y) \in gPp\})) \setminus slc'$
$slc \ := (slc' \cup \{x \mid (x,y) \in gPp\}) \setminus dots$
RETURN $(slc, dots)$
END

Abbildung 20: **analyseGoto** $(f, t, slc, dots)$

Die Funktion **analyseGoto** in Abbildung 20 ermittelt als erstes alle Paragraphen (gPp), welche durch die Sprunganweisung zur Anwendung kommen. Jeder dieser Paragraphen wird sodann mit Hilfe der Funktion analyseParts gesondert analysiert und mit der Funktion `joinStates` zu einem Zwischenergebnis (slc'', $dots''$) kumuliert.

Die Programmzeilen der Sprunganweisung ($\texttt{fill}(f,t)$) werden auch dann in das Program-Slice aufgenommen, wenn die untersuchten Paragraphen keinen Beitrag liefern sollten. Der Grund dafür ist klar: falls nämlich der Kontrollfluss des Programms bis zur fraglichen Sprunganweisung gelangt, wird er unweigerlich zu einer anderen Stelle im Programm gelenkt. Mit anderen Worten, diese Sprunganweisung wird auf jeden Fall aktiviert.

Daraus folgt weiter, dass zusätzlich auch die Sprungmarken (Paragraphennamen) in das Program-Slice integriert werden müssen, um den Erhalt eines syntaktisch korrekten Program-Slices zu gewährleisten.

Die Funktion **analyseAtom** in Abbildung 21 analysiert eine Sequenz von einfachen Anweisungen und arbeitet dabei wiederum vom Ende zum Anfang. Das resultierende Zwischenergebnis (slc) wird dem jeweiligen Vorgänger weitergereicht. Mit der Funktion analyseAtom liesse sich die Sequenz grundsätzlich auch als Ganzes analysieren. Allerdings

```
analyseAtom (f, t, slc, dots)
  BEGIN
    ls   := fill(f, t)
    WHILE ls ≠ ∅ DO
      slc := transitiveClosureVar(max(ls), max(ls), slc)
      ls  := ls \ {max(ls)}
    END
    dots  := dots \ slc
    RETURN (slc, dots)
  END
```

Abbildung 21: **analyseAtom** $(f, t, slc, dots)$

bestünde dann die Gefahr, dass dabei auch solche für das Slice irrelevante Anweisungen übernommen würden. Das Beispiel in Abbildung 22 mag als Illustration dienen: Gesetzt den Fall, unmittelbar vor der Analyse dieser Anweisungen sei nur die Variable `x` im bis dahin berechneten Program-Slice enthalten. Würden nun alle Anweisungen gemeinsam in die Berechnung einbezogen, so müssten folglich alle in das Slice integriert werden. Wird aber jede Anweisung einzeln von hinten nach vorne analysiert, wird nur die erste Anweisung `MOVE a TO x` in das Slice übernommen.

```
MOVE a TO x
MOVE b TO a
MOVE c TO b
```

Abbildung 22: Einfache Anweisungen in einem ATOM

Durch diesen verbesserten dritten Ansatz haben wir erreicht, dass der Untersuchungsbereich auf eine Zeile reduziert werden kann. Mit anderen Worten, die Funktion transitiveClosureVar (siehe Abbildung 7) wird die transitive Hülle der Abhängigkeiten eines Slicing-Kriteriums bei jedem Aufruf jeweils höchstens noch um eine Zeile erweitern. Unter Berücksichtigung der Regel 1 in Abschnitt 2.2 (jede Anweisung gesondert in einer separaten Zeile) lässt sich nun foldern, dass das hier vorgestellte syntax-basierte Program-Slicing jede einzelne Anweisung gesondert analysieren kann.

6 Evaluation

6.1 Eigenschaften der syntax-basierten Slices

Die Eigenschaften der syntax-basierten Slices werden massgeblich durch die Funktion transitiveClosureVar definiert, denn sie berechnet die transitive Hülle der Abhängigkeiten des Slicing-Kriteriums im Untersuchungsbereich. War der Untersuchungsbereich in den ersten beiden Ansätzen noch identisch mit dem Betrachtungbereich, so ist er im verbesserten Ansatz kleiner als der Betrachtungbereich. Dies wird zwangsläufig zu unterschiedlichen Eigenschaften der berechneten Program-Slices führen, welche wir in den folgenden Abschnitten beschreiben werden.

6.1.1 Die ersten beiden Ansätze

1. *Modus der Analyse:*

   Mit den beiden ersten Ansätzen können nur statische Slices berechnet werden. Bei einer dynamischen Analyse müssten nämlich auch die Inhalte von Variablen in die Berechnung der Slices einbezogen werden. Auch ist es selbst für eine abstrakte Interpretation von Anweisungen erforderlich, dass die interne Repräsentation des Programmtexts zur Verfügung steht, was aber im vorliegenden Falle nicht gegeben ist.

2. *Ausführbarkeit:*

   Bei dem hier generierten Program-Slice ist die Ausführbarkeit gewährleistet, da die berechnete transitive Hülle nachträglich syntaktisch vervollständigt wird.

3. *Richtung der Analyse:*

   Die Funktion transitiveClosureVar liefert die transitive Hülle der Abhängigkeiten zum interessierenden Slicing-Kriterium,wobei der Untersuchungsbereich richtungsunabhängig als Ganzes analysiert wird. Aus diesem Grund müssen die berechneten Program-Slices sowohl als rückwärts- als auch als vorwärts-orientiert betrachtet werden. Sie werden deshalb in der Regel eher grossen Umfang aufweisen und ergo weniger präzise sein in ihrer Aussage über ein bestimmtes Slicing-Kriterium als jene, welche durch Ansätze berechnet werden, die nur rückwärts- resp. vorwärts-orientierte Slices liefern.

4. *Weitere Merkmale:*

   Da das Program-Slice, wie er durch den ersten Ansatz berechnet wird, immer alle Sprunganweisungen und Subroutinenaufrufe enthält, ist die untere Grenze für den Umfang eines Slices bestimmt. Er besteht somit aus jenen Zeilen, welche

   (a) Sprunganweisungen und Subroutinenaufrufe des Programms enthalten
   (b) durch die syntaktische Ergänzung integriert werden
   (c) in der transitiven Hülle der Abhängigkeiten aller verwendeten Variablen innerhalb von Konstrukten, welche in das Slice aufgenommen wurden, vorkommen (z.B. in Bedingungen von Schleifen oder bedingten Anweisungen)

   Dieses Gerüst ist für jedes Slicing-Kriterium identisch. Mit anderen Worten, es ist in *allen* durch den ersten Ansatz berechenbaren Program-Slices enthalten.

6.1.2 Der verbesserte Ansatz

1. *Modus der Analyse:*

   Auch hier kommt, wie bei den ersten beiden Ansätzen, die statische Analyse zur Anwendung (siehe gleichnamiger Punkt in Abschnitt 6.1.1).

2. *Ausführbarkeit:*

   Die hier berechneten Program-Slices sind ebenfalls ausführbar, wie bei den ersten beiden Ansätzen (siehe gleichnamiger Punkt in Abschnitt 6.1.1).

3. *Richtung der Analyse:*

   Die Analyserichtung bezeichnen wir als *quasi-rückwärts*, d.h. sie ist generell rückwärts-orientiert, aber nicht ausschliesslich. Die Programmteile (z.B. Parts, Chunks, Blocks) werden zwar wie bei einer gängigen rückwärts-orientierten Analyse durchlaufen, aber innerhalb der kleinsten Untersuchungseinheiten (Atome) wird jede Anweisung richtungs-unabhängig analysiert. Das kommt daher, dass die Funktion transitiveClosureVar den Untersuchungsbereich (hier genau eine Zeile) als Ganzes betrachtet, ohne Rücksicht auf die Richtung.

4. *Weitere Merkmale:*

   Bei der Berechnung des Program-Slices wächst sein Umfang kontinuierlich an, denn alle einmal in das Slice aufgenommenen Zeilen und mit ihnen auch die Variablen, bleiben darin erhalten. Mit anderen Worten, die Slices können auch Sequenzen von Anweisungen enthalten, deren direkte oder indirekte Wirkung überschrieben werden und darum auf das Slicing-Kriterium im berechneten Slice keinen Einfluss mehr ausüben können.

   ```
   MOVE a TO x
   MOVE b TO x
   ```

   Abbildung 23: Überschreiben der Wirkung von Anweisungen

   Das Beispiel in Abbildung 23 verdeutlicht das Problem: Obwohl die erste Anweisung ohne Wirkung bleibt, weil bereits in der nächsten Anweisung der Wert von `x` unabhängig von der ersten neu beschrieben (definiert) wird, wird die für das Program-Slice irrelevante Anweisung nicht eliminiert. Der Grund dafür ist in unseren Voraussetzungen zu suchen: Da wir aus den Informationen eines Cross-Referencers nicht bestimmen können, welche der Variablen auf einer Zeile beschrieben resp. nur gelesen wird, müssen wir im vorliegenden Fall annehmen, dass beide Variablen, `a` und `x`, beschrieben werden, wodurch es untersagt ist, diese Anweisung aus dem Slice zu entfernen.

6.2 Diskussion

In allen vorgestellten Ansätzen werden ausschliesslich Informationen verwendet, die aus der PROCEDURE DIVISION stammen. Abhängigkeiten, welche in der DATA DIVISION enthalten sind, werden vernachlässigt:

- Aliasing durch RENAMES und REDEFINES werden nicht berücksichtigt. Durch die beiden genannten Schlüsselwörter wird ein bestimmter Speicherplatz mehreren Variablen gemeinsam zugewiesen. Wird eine dieser Variablen modifiziert, werden die anderen Variablen ebenfalls davon betroffen. Die Variablen stehen also in einer echten, gegenseitigen Abhängigkeit.

- Die Hierarchie der Felder (wird in COBOL durch unterschiedliche Level-Nummern ausgedrückt) wird nicht genutzt. Modifikationen von Variablen wirken sich auf einige, u.U. sogar auf alle ihre Sub-Felder aus. Umgekehrt beeinflussen Modifikationen in Sub-Feldern alle Variablen, welche dieses Feld als Teil enthalten.

Könnten auch solche Seiteneffekte von Modifikationen aufgedeckt werden, liessen sich diese zusätzlichen Beziehungen ebenfalls in das Program-Slicing einbeziehen.

Die in Abschnitt 2.2 für die Gliederung der zu analysierenden Programmtexte aufgestellten Regeln wirken einigen Fehlerquellen erfolgreich entgegen. Befinden sich z.B. auf der selben Zeile zwei gleichnamige Objekte (Schlüsselworte, Variablennamen, ...), so kann dies zu Informationsverlusten führen, da ein Cross-Referencer in der Regel für ein Objekt aus dem Programmtext jede Zeilennummer *höchstens* einmal vergibt.

Eine weitere Schwierigkeit betrifft die Variablen innerhalb von Bedingungen (z.B. in Schleifen oder bedingten Anweisungen). Diese Variablen werden nämlich in allen drei Ansätzen genau so behandelt wie Variablen in einfachen Anweisungen; mit anderen Worten, sie werden als voneinander abhängig betrachtet, obwohl es offensichtlich ist, das sie nur gelesen werden. Der Umfang von Program-Slices könnte also beträchtlich reduziert werden, wenn es gelingt die Variablen in den Bedingungen nur als gelesene zu behandeln.

Bei unserem dritten Vorschlag für das syntax-basierte Program-Slicing stellt sich das Problem, dass die Analyse in bestimmten Fällen nicht terminiert. Diese seltenen Fälle sind auf die Behandlung der Subroutinenaufrufe (PERFORM ... THRU) zurückzuführen. Wie wir uns erinnern, wird in diesem Ansatz ein Paragraph rückwärts von hinten nach vorne analysiert. Trifft die Analyse nun auf einen Subroutinenaufruf, so bearbeitet sie diesen zunächst, bevor sie zu seinem Vorgänger weiterschreitet. Falls nun der momentan analysierte Paragraph innerhalb der durch den Subroutinenaufruf aktivierten Sequenz von Paragraphen liegt, so wird die Analyse nie terminieren können. Solche Problemfälle werden erkannt, und die Berechnung wird mit dem Hinweis der Nicht-Termination abgebrochen.

7 Zusammenfassung und Ausblick

Im vorliegenden Bericht haben wir aufgezeigt, dass ein Program-Slicing möglich ist, welches sich ausschliesslich auf die grobkörnige (weil zeilen-orientierte) Information aus den Cross-Referencer-Daten abstützt. Die Motivation, die dahinter steht, war und ist, zu zeigen, dass

1. selbst mit Hilfe so rudimentärer Informationen, wie sie ein Cross-Referencer liefert, ein Program-Slicing durchgeführt werden kann, und

2. das Program-Slicing auch auf real-world Applikationen, welche in COBOL geschrieben sind, angewandt werden kann, um das Verstehen der Funktionalität eines Programms zu unterstützen.

Wir verweisen in diesem Zusammenhang auf [Ven95], wo Venkatesh das Slicing für reale C–Programme untersucht und dabei betont, dass sich das Program-Slicing in der Praxis bewähren müsse und nicht nur an akademischen Problemen gezeigt werden soll.

Die vorgestellten Ansätze arbeiten bewusst ausschliesslich mit Daten, die von gängigen COBOL-Compilern bereitgestellt werden können. Damit lässt sich das Problem umgehen, für die Abhängigkeiten von Programmgrössen (z.B. Variablen) erst eine geeignete interne Darstellung zu generieren, welche für COBOL äussert schwierig ist. Ansätze dazu existieren zwar bereits ([Fae95]), aber kommen in der Praxis noch nicht zum Einsatz.

Alle drei beschriebenen Ansätze wurden in Standard ML implementiert, einer modernen funktionalen Programmiersprache mit Type-Checking ([MTH90, Pau91]). Die nach den beiden ersten Ansätzen generierten Slices sind sowohl vorwärts- als auch rückwärts-orientiert. Mit anderen Worten, sie enthalten alle Effekte, die auf ein Slicing-Kriterium wirken, und gleichzeitig aber auch jene Effekte, die das Slicing-Kriterium seinerseits auf nachfolgende Anweisungen haben kann. Es liegt in der Natur der Sache, dass diese Slices voluminöser ausfallen als unter Berücksichtigung einer Analyserichtung.

Der dritte Ansatz führt darum das Kriterium der Analyserichtung ein. Wir bezeichnen die berechneten Slices als quasi-rückwärts, da bei der Analyse eine Sequenz von Anweisungen zwar grundsätzlich von hinten nach vorne bearbeitet, die einzelnen Anweisung dennoch richtungs-unabhängig betrachtet werden. Eine weitere Verbesserung leistet die Reduktion des Untersuchungsbereichs, mit welchem bei der Analyse gearbeitet wird. Arbeiten die beiden ersten Ansätze noch mit dem gesamten Programm resp. einem Paragraphen, so umfasst der Untersuchungsbereich beim dritten Ansatz noch genau eine

Anweisung. Damit kann der Umfang der generierten Program-Slices weiter verringert werden, was einer Erhöhung der Aussagekraft gleichkommt.

Dieser Gewinn hat aber seinen Preis. Alle Ansätze basieren auf den zeilen-orientierten Informationen eines Cross-Referencers, weshalb wir für die Gestaltung des Texts Regeln einführen mussten. Nur wenn diese befolgt, oder durch einen Pretty-Printer sichergestellt werden, sind die Aussagen und damit die Qualität eines generierten Slices optimal.

Die Resultate können noch weiter optimiert werden, indem nur solche Sprunganweisungen und Subroutinenaufrufe in das Program-Slice aufgenommen werden, die relevant sind. Liegt das Slicing-Kriterium beispielsweise innerhalb einer bedingten Anweisungen, so wäre nur ein Ast für das Slicing relevant, während die Sprunganweisungen und Subroutinenaufrufe im anderen Ast vernachlässigt werden könnten.

Wir haben den Betrachtungsbereich für das Program-Slicing im zweiten und dritten Ansatz auf einen Paragraphen eingeschränkt, weil der Kontrollfluss zwischen den Paragraphen nicht ohne weiteres aus den Informationen eines Cross-Referencers bestimmt werden kann. Bei einer Weiterentwicklung unserer Ansätze wäre es eine vordringliche Aufgabe, auch diese Beziehungen im Program-Slicing zu berücksichtigen, wozu ein Call-Graph, der allerdings vorgängig generiert werden müsste, ein vielversprechender Ansatz darstellen könnte.

Literatur

[ASU88] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilerbau.* Addison-Wesley, 1988.

[BFK+93] P. Baumann, J. Fässler, M. Kiser, Z. Öztürk, and L. Richter. Using Standard ML in Software Maintenance: A Promising Approach. Technical Report 93.37, Department of Computer Science, University of Zurich, Switzerland, September 1993.

[BFK+94] P. Baumann, J. Fässler, M. Kiser, Z. Öztürk, and L. Richter. Semantics-based Reverse Engineering. Technical Report 94.08, Department of Computer Science, University of Zurich, Switzerland, July 1994.

[BFKO93a] P. Baumann, J. Fässler, M. Kiser, and Z. Öztürk. Beauty and the Beast or A Formal Description of the Contol Constructs of Cobol and its Implementation. Technical Report 93.39, Department of Computer Science, University of Zurich, Switzerland, September 1993.

[BFKO93b] P. Baumann, J. Fässler, M. Kiser, and Z. Öztürk. Using Functional Parsing to Achieve Quality in Software Maintenance. Technical Report 93.36, Department of Computer Science, University of Zurich, Switzerland, September 1993.

[Fae95] J. Faessler. Entwurf und Implementation des funktionalen Parser-Generators CoP. Technical Report 95.27, Institut für Informatik der Universität Zürich, October 1995.

[FKO92] J. Fässler, M. Kiser, and Z. Öztürk. AEMES - First Specification. internal report KWF2326.1(AEMES)-D01-SPEC V1.0 (12.92), Institut fuer Informatik, University of Zurich, Winterthurerstrasse 190, 8021 Zurich, December 1992.

[FOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Hau89] P.A. Hausler. Denotational Program Slicing. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, volume 2, pages 486–494, 1989.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[Kis94] M. Kiser. The Wireable User Interface - A Graphical Tool Integration Mechanism. Technical Report 94.09, Institut für Informatik der Universität Zürich, July 1994.

[KL88] Bogdan Korel and Janusz Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Oez95] Z. Oeztürk. Semantik-basiertes Program-Slicing für Sprachen mit Komplexen Kontrollstrukturen. Technical Report 95.22, Institut für Informatik der Universität Zürich, June 1995.

[OO84] K.J. Ottenstein and L.M. Ottenstein. The Program Dependence Graph in a Software Development Environment. *ACM SIGSOFT Software Engineering Notes*, 9(3):177–184, May 1984.

[Pau91] L.C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[Ven91] G.A. Venkatesh. The Semantic Approch to Program Slicing. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26 (6), pages 107–119. ACM Press, June 1991.

[Ven95] G.A. Venkatesh. Experimental Results from Dynamic Slicing of C Programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–216, March 1995.

[Wei82] M. Weiser. Programmers use Slices when Debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[Wei84] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.