Semantik-basiertes Program-Slicing für Sprachen mit komplexen Kontrollstrukturen**

(IFI TR 95.22)

Zafer Öztürk *

Swiss Life, General Guisan-Quai 40, 8022 Zürich, Schweiz
Email: zafer@swssai.uu.ch
Telefon: +41-1-711 3707. Fax: +41-1-711 5007.

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, Denotationelle Semantik, Continuation-Semantics, Standard ML, COBOL

Inhalt: Program-Slices wurden in den mittleren 80er Jahren durch M. Weiser [Wei84] eingeführt. Seither hat man unterschiedliche Ansätze vorgeschlagen, wie sie aus dem Programmtext hergeleitet werden können. Anfangs der 90er Jahre wurde erstmals die denotationelle Semantik verwendet, um Program-Slicer zu beschreiben. Als Zielsprache dienten einfache While-Sprachen ohne `GOTO`-Anweisung und ohne Prozeduren. Im Projekt AEMES[1] hingegen wird nun mit einer Zielsprache gearbeitet, die gerade mit diesen Sprüngen und Prozeduren operiert. Um das Program-Slicing auch für solche Sprachklassen beschreiben zu können, muss der Kontrollfluss innerhalb eines Programms berücksichtigt werden. Damit der Program-Slicer trotzdem mit Hilfe der denotationellen Semantik beschrieben werden kann, setzen wir die Continuation-Semantics ein. Damit werden Prozeduren und Sprünge im Kontrollfluss erfassbar und manipulierbar.

Die entsprechenden Konzepte werden wir an der Sprache *MiCo*[2] mit `GOTO`-Anweisung und Prozeduren zeigen. Anschliessend an diese Beschreibung mit Hilfe der Continuation-Semantics werden wir einen Vorschlag für die Abschätzung des Berechnungsaufwandes für die Program-Slices durch unseren Ansatz präsentieren.

**Der vorliegende Bericht erscheint in identischer Form im Juni '95 auch als Bericht 10/95 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 die Bull(Schweiz).

[2] *MiCo* steht für *Mi*ni *Co*bol und kann als Kernsprache von COBOL betrachtet werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Im Rahmen des Projekts AEMES ([BFKO93b, BFK+93, BFKO93a, BFK+94, FKO92]) werden Werkzeuge zur Analyse von COBOL-Programmen entwickelt. Da die untersuchten Programmtexte in der Regel sehr umfangreich sind (mehrere 1000 Programmzeilen), werden Methoden und Werkzeuge gesucht, die kompaktere Sichten auf den Programmtext erlauben. In AEMES werden zwei Klassen von Werkzeugen unterschieden. Die erste Klasse umfasst alle Werkzeuge, welche ausschliesslich syntaktische Eigenschaften des Programmtexts verwenden. Dazu gehören beispielsweise Program-Browser und Cross-Referencer. Die zweite Klasse umfasst diejenigen Werkzeuge, welche zusätzlich die semantischen Eigenschaften des Programmtexts ausnutzen, um die gewünschten Informationen zu extrahieren. In letztere fallen unter anderem auch Program-Slicer und Daten-/Kontrollfluss-Analysatoren.

Im vorliegenden Bericht wird ein Program-Slicer vorgestellt, der auf der denotationellen Semantik der zu analysierenden Programmiersprache basiert. Der Program-Slicer selber wird ebenfalls mit Hilfe der denotationellen Semantik beschrieben.

Der Begriff *Program-Slice* wurde erstmals von M. Weiser [Wei82, Wei84] eingeführt. Seine Untersuchungen haben gezeigt, dass Wartungsprogrammierer beim Versuch ein Programm zu verstehen, dieses mental in Teilprogramme gliedern, wovon jedes für sich einen gewissen Aspekt des Originalprogramms abbildet. Das klassische Beispiel eines Program-Slices enthält diejenigen Programmteile, welche auf eine zu untersuchende Variable an einer bestimmten Programmstelle Einfluss nehmen können (oft verwendet im Debugging).

Um Program-Slices aus einem Programmtext abzuleiten, gibt es verschiedene Ansätze. Diese basieren zum einen auf dem Programm-Ablaufpfad, wie er bereits von M. Weiser vorgeschlagen wurde ([Wei84, LR87]), zum andern auf den Abhängigkeitsgraphen [OO84, FOW87, HRB90, AH90, HR92], welche aus dem Programm generiert werden oder aber auf der denotationellen Semantik einer Programmiersprache [Hau89, Ven91]. Unser Interesse gilt im folgenden vor allem denjenigen Ansätzen, bei welchen die denotationelle Semantik verwendet wird.

Hier ist zunächst der Ansatz von P.A. Hausler [Hau89] zu erwähnen, welcher für eine bestimmte Variable den zugehörigen Program-Slice generiert. Seine Zielsprache ist eine strukturierte While-Sprache, d.h. sie enthält keinen Sprungbefehl und auch keine Prozeduren. Unter Verwendung einer vergleichbaren Zielsprache wurde ein weiterer Ansatz von G.A. Venkatesh in [Ven91] vorgeschlagen. Im Gegensatz zu P.A. Hausler werden hier neu die Slices für sämtliche Variablen in einem einzigen Analysedurchgang erzeugt. In beiden Ansätzen kommt die denotationelle Semantik zur Anwendung, um sowohl die Semantik der Zielsprache als auch das Program-Slicing selbst zu beschreiben.

Dies gilt, wie oben bereits erwähnt, auch für den im folgenden vorgestellten Ansatz. Allerdings werden hier die Ansätze von P.A. Hausler und G.A. Venkatesh insofern erweitert, als damit auch Programmiersprachen mit Sprüngen und Prozeduren ohne Parameter einem semantikbasierten Program-Slicing zugänglich werden. Auch wir werden zur Beschreibung des Program-Slicers die denotationelle Semantik verwenden. Allerdings ist für unseren Ansatz die von P.A. Hausler und G.A. Venkatesh gewählte Direct-Style Semantics ([AS88, NN92, Sch86]) nicht mehr ausreichend, weshalb wir die Continuation-Style Semantics anwenden (siehe Abschnitt 2).

Bei der Beschreibung der semantischen Funktionen des Program-Slicers werden wir die Notation von Standard ML verwenden [MTH90]. Standard ML ist eine funktionale Programmiersprache mit strikter Evaluation und strengem Type-Checking. Das Pattern-Matching von Standard

ML unterstützt das Implementieren von semantischen Funktionen, da sich damit die semantischen Gleichungen sehr einfach in die Standard ML Notation übertragen lassen.

Der vorliegende Bericht gliedert sich in folgende Teile: in Abschnitt 2 werden einschlägige Begriffe aus der Semantik kurz erläutert und der von uns vorgeschlagene Algorithmus in groben Zügen skizziert. Abschnitt 3 beschreibt die Grammatik und Besonderheiten unserer Zielsprache *MiCo*, welche mitunter auch einen Sprungbefehl (`GOTO`) und Prozeduraufrufe (`PERFORM`) enthält. In Abschnitt 4 wird der Ansatz an der Sprache *MiCo* demonstriert. Hierbei werden nebst dem abstrakten Syntaxbaum auch die semantischen Bereiche, die Hilfsfunktionen und die semantischen Funktionen für die einzelnen syntaktischen Sprachkonstrukte definiert. In Abschnitt 5 werden wir die Zeitkomplexität unseres Ansatzes bestimmen. Mit einer Zusammenfassung und dem Ausblick schliessen wir in Abschnitt 6.

2 Program-Slicing mit der Continuation-Semantics

2.1 Program-Slices

Den Begriff des Program-Slicing hat, wie oben bereits erwähnt, M. Weiser geprägt und auch eine Beschreibung dazu gegeben:

> Bei einer gegebenen Menge v von Variablen und einer Programmstelle p, hat das Program-Slicing das Ziel, eine Projektion des Originalprogramms zu erzeugen (durch eliminieren von Anweisungen), so dass der resultierende Program-Slice und das Originalprogramm für die Variablen in v an der Programmstelle p die gleichen Werte berechnen.

Es wird implizit angenommen, dass der Program-Slice so klein wie möglich gemacht werden soll. Das Paar (p, v) wird Slicing-Kriterium genannt. Die Program-Slices können anhand folgender Eigenschaften charakterisiert werden:

- *statisch* $\leftrightarrow$ *dynamisch:* Statisch heisst, dass der Slice für alle Anfangswerte der Variablen gilt, im Gegensatz zu dynamisch, wo der Slice nur für eine ganz bestimmte Variablenbelegung gültig ist.

- *vorwärts* $\leftrightarrow$ *rückwärts:* Rückwärts orientierte Slices erfassen alle Anweisungen, die auf bestimmte Variablen an einer Programmstelle einen Einfluss ausüben können. Die vorwärts orientierten Slices beinhalten alle Anweisungen, die von bestimmten Variablen von einer Programmstelle aus beeinflusst werden können.

- *ausführbar* $\leftrightarrow$ *nicht-ausführbar:* Ein Slice heisst ausführbar, falls er für die interessierenden Variablen das gleiche Verhalten zeigt wie das Originalprogramm. Ein nicht-ausführbarer Slice erfasst nur die Datenabhängigkeiten von Variablen. Da der Kontrollfluss vernachlässigt wird, kann es sein, dass die Resultate der interessierenden Variablen von denen des Originalprogramms abweichen.

Eine Gruppierung in verschiedene Klassen anhand der oben erwähnten Kriterien und ihre semantische Beschreibung wird in [Ven91] gegeben.

2.2 Denotationelle Semantik

In der Einleitung sagten wir, dass in unserem Ansatz die denotationelle Semantik zur Anwendung kommt, auf die wir hier kurz näher eingehen wollen. Für weitergehende Ausführungen verwiesen wir auf [AS88, NN92, Sch86].

In der denotationellen Semantik werden die Effekte eines Programms beschrieben, und nicht wie das Programm ausgeführt wird. Mit ihrer Hilfe lässt sich die Semantik einer Sprache sehr einfach und präzise beschreiben. Die folgenden Eigenschaften zeichnen die denotationelle Semantik aus:

- zu jeder syntaktischen Kategorie existiert eine semantische Funktion.
- zu jedem Konstrukt in einer syntaktischen Kategorie existiert eine semantische Gleichung, welche seine Semantik ausschliesslich aus den Bedeutungen seiner direkten Komponenten definiert.

Aus der letzten Eigenschaft leitet sich das Hauptmerkmal der denotationellen Semantik ab. Zur Beschreibung der Semantik eines syntaktischen Elements dürfen nur die direkten Komponenten verwendet werden, aber nicht das syntaktische Element selbst. Somit kann eine Schleife nicht mehr als eine Kombination einer Bedingung und derselben Schleife dargestellt werden. Aus diesem Grund müssen semantische Gleichungen für Schleifen mit Hilfe einer Fixpunktfunktion definiert werden.

2.3 Continuations in der denotationellen Semantik

Für strukturierte Sprachen ohne Prozeduren und Sprünge ist die einfache Direct-Style Semantik durchaus hinreichend. Der Kontrollfluss des Programms ist implizit durch die Reihenfolge der syntaktischen Elemente gegeben. Er muss deshalb nicht explizit dargestellt werden. Dies hat aber zur Folge, dass er nicht manipuliert werden kann.

Um nun auch Prozeduren und Sprünge, d.h. den Kontrollfluss eines Programms, erstens besser darstellen und zweitens manipulieren zu können, verwendet man in der denotationellen Semantik die Continuation-Style Semantik. Eine Continuation ist eine Funktion, welche die Effekte aller Anweisungen von einer bestimmten Stelle im Programm aus bis zu dessen Ende enthält. Wird dieser Continuation ein Zustand übergeben, liefert sie direkt den Effekt des Restprogramms auf diesen Zustand.

Die Continuation für das gesamte Programm wird wie folgt zusammengebaut: Jede Anweisung muss ihre Continuation um ihre eigenen Effekte ergänzen und diese neue Continuation ihrem Vorgänger zur Verfügung stellen. Die letzte Anweisung des Programms, welche ja keinen eigentlichen Nachfolger hat, bekommt als Eingabe eine intiale Continuation.

Ausführlich beschrieben und mit Beispielen versehen, findet sich das hier skizzierte Konzept der Continuations in [AS88, NN92, Sch86].

2.4 Algorithmus

Wir wollen hier unseren Ansatz zunächst in Worten skizzieren. Die Ausführungen in den folgenden Abschnitten werden dadurch einfacher nachzuvollziehbar. Das Resultat des hier vorgestellten Ansatzes liefert statische, rückwärts-orientierte und ausführbare Program-Slices. Die interessierende Programmstelle ist immer das Ende des Programms. Damit reduziert sich das Slicing-Kriterium auf den Namen einer Variablen, da die Programmstelle implizit die letzte Zeilennummer ist. Jeder Variablen wird ihr Program-Slice zugeordnet. Dieser wird als eine Menge von Zeilennummern dargestellt, welche den Zeilennummern aus dem Originalprogramm entsprechen.

Zu Beginn werden die Inhalte der Variablen mit der leeren Liste initialisiert (entspricht einem leeren Program-Slice). Anschliessend wird das Programm Anweisung um Anweisung analysiert, wobei die Reihenfolge der bearbeiteten Anweisungen durch den Kontrollfluss bestimmt wird.

- Bei Goto-Anweisungen wird mit der Analyse dort weitergefahren, wo die Sprungadresse hinweist.
- Bei Prozeduraufrufen werden zuerst die Anweisungen in der Prozedur bearbeitet und anschliessend direkt nach dem Prozeduraufruf weitergefahren.

Bei Schleifen und bedingten Anweisungen ist zu berücksichtigen, dass die Anweisungen innerhalb des Schleifenkörpers, resp. des linken und rechten Astes der bedingten Anweisung, in einer Kontrollabhängigkeit mit den Variablen in der Bedingung stehen. Deshalb ist es erforderlich, dass sie ebenfalls in den Slice integriert werden.

Bei Zuweisungen zu einer Variablen wird jeweils der betreffende Slice aktualisert. Die neu berechnete Variable steht in einer Datenabhängigkeit von den im Ausdruck verwendeten Variablen. Der neue Slice ist die Vereinigung von drei Mengen:

1. $\{$*aktuelle Zeilennummer*$\}$
2. $\bigcup$ (*zugehöriger Slice der Variablen im berechneten Ausdruck*)
3. $\bigcup$ (*zugehöriger Slice der Variablen, zu denen die neu berechnete Variable in einer Kontrollabhängigkeit steht*)

Dieses Vorgehen hat zwei grosse Vorteile: Erstens wird mit dem Vereinen der Slices von Variablen direkt auch die transitive Hülle der Abhängigkeiten gebildet. Zweitens werden mit einem einzigen Analysedurchgang durch den Programmtext die Program-Slices zu allen Variablen gleichzeitig erzeugt.

Bei unserem Ansatz ist der Aufwand für das Bilden der Slices etwas höher (siehe Abschnitt 5), als wenn nur ein bestimmter Slice (vergl. [Hau89]) generiert werden müsste. Dafür wird aber das interaktive Benutzen entschieden effizienter, da die Program-Slices der einzelnen Variablen lediglich abgefragt, nicht aber generiert werden müssen.

3 Die Sprache *MiCo*

3.1 Motivation

Wie in der Einleitung erwähnt, ist COBOL die Zielsprache im Projekt AEMES. Die dort entwickelten Werkzeuge, mitunter auch der Program-Slicer, müssen in der Lage sein, mit den Besonderheiten dieser Sprache umzugehen.

Je nach Version und Anbieter weist die Sprache COBOL zwischen 250 und 400 Schlüsselwörter auf. Zusätzlich haben die meisten Sprachkonstrukte mehrere Varianten und sind z.T. redundant untereinander. In Abb. 1 sind Programmausschnitte aufgelistet, die alle den gleichen Effekt haben, nämlich dass die Werte der Felder `x`, `y` und `z` addiert werden und den beiden Feldern `s` und `t` zugewiesen werden.

```
ADD x, y, z TO s
ADD x, y, z TO t
```

oder

```
ADD x, y, z GIVING s, t
```

oder

```
COMPUTE s, t = (x + y + z)
```

Abbildung 1: Varianten zur Addition von Zahlen in COBOL

Für die Evaluation von Analyseansätzen war es notwendig, den enormen Sprachumfang und Variantenreichtum von COBOL zu reduzieren. Aus diesem Grund wurde die Sprache *MiCo* definiert. Hierbei wurde besonders darauf geachtet, die erwähnten Redundanzen zwischen den Sprachkonstrukten in COBOL zu vermeiden. *MiCo* enthält auch die meisten Kontrollkonstrukte, insbesondere das `GOTO` und das `PERFORM`, und weitere Besonderheiten von COBOL (z.B. Aliasing, keine lokalen Variablen). Der bei *MiCo* stark verminderte Umfang an syntaktischen Strukturen hat zusätzlich den Vorteil, dass sich ihre Semantik kompakt und vollständig beschreiben lässt (siehe in [BFKO93a]).

In den nachfolgenden Abschnitten werden wir zunächst die Grammatik von *MiCo* beschreiben, anschliessend einige Besonderheiten der Sprache skizzieren, um schlussendlich anhand eines *MiCo*-Programms die Syntax zu veranschaulichen und die informelle Semantik dazu vorzustellen.

3.2 Die Grammatik

Die angesprochenen Vereinfachungen in *MiCo* betreffen sowohl die Grobstruktur als auch die Feinstruktur von COBOL (siehe Abb. 2 und 3).

Auf der Stufe der Divisions interessieren wir uns ausschliesslich für die `DATA DIVISION` und die `PROCEDURE DIVISION`. Wie auch in COBOL sind die in der `DATA DIVISION` definierten Variablen alle global. Da wir uns auf numerische Operationen beschränken wollen, sind alle Variablen vom Type `NUMERIC`. Das Überlappen von Feldern in COBOL wird in *MiCo* durch das `ALIAS`-Konstrukt nachgebildet, welches uns erlaubt, mehrere Variablennamen auf dieselbe Speicheradresse abzubilden.

In der `PROCEDURE DIVISION` haben wir die Zwischenstrukturen `SECTION` und `SENTENCE` weggelassen, da ihre Funktionalität auch durch `PARAGRAPH` resp. durch `STATEMENT` realisiert werden können.

Die Vereinfachungen in der Feinstruktur von *MiCo* betreffen insbesondere die Anweisungen (Statement). Bei der Definition galt unser Interesse vor allem den Kontrollstrukturen in COBOL, die bei allen Analyseverfahren eine entscheidende Rolle spielen. Dazu gehören der Subroutinenaufruf `PERFORM`, die unbedingte Sprunganweisung `GOTO` und die bedingte Anweisung `IF-THEN-ELSE`. Die `SKIP`-Anweisung ermöglicht es, dass auch `IF-THEN` dargestellt werden können. Zusätzlich wurde eine `MOVE`-Anweisung eingeführt, um Manipulationen an Variablen eines *MiCo*-Programms vornehmen zu können. Diese `MOVE`-Anweisung ist gegenüber der COBOL-Variante erweitert, so dass auf der linken Seite des TO statt nur Variablen auch ganze Ausdrücke

Program	→	"PROGRAM" Identifier "DATA" "DIVISION" DeclarationSeq "END" "PROCEDURE" "DIVISION" ParagraphSeq "END" "END"
DeclarationSeq	→	Declaration { Declaration }
Declaration	→	VariableSeq ":" Type
VariableSeq	→	Variable { "," Variable }
Variable	→	Identifier { "ALIAS" Identifier }
Type	→	"NUMERIC"
ParagraphSeq	→	Paragraph { Paragraph }
Paragraph	→	"PARAGRAPH" Identifier StatementSeq "END"
StatementSeq	→	Statement { "." Statement }

Abbildung 2: Grobstrukturen von *MiCo*

benutzt werden dürfen. Damit lässt sich die in anderen Sprachen übliche Zuweisung darstellen. Die Syntax von *MiCo* wurde bewusst einfach gehalten und so gewählt, dass damit das Wesentliche eines Analyseansatzes (z.B. Program-Slicing) untersucht werden konnte. Im folgenden Abschnitt werden einige Besonderheiten von *MiCo* resp. COBOL näher beschrieben.

3.3 Besondere Merkmale

Nachdem im vorangehenden Abschnitt die Syntax von *MiCo* beschrieben wurde, wollen wir im nun folgenden Abschnitt einige Eigenheiten von *MiCo* aufzeigen, die ihren Ursprung in COBOL haben.

Zunächst ist die implizite resp. explizite Reihenfolge der ausgeführten Paragraphen zu nennen. Die erste ist bekannt unter dem Namen Fall-Through und bedeutet, dass die Paragraphen sequentiell in syntaktischer Ordnung durchlaufen werden. Die zweite wird durch den Programmierer festgelegt. Er verwendet dazu die Kontrollkonstrukte `GOTO` und `PERFORM`. Paragraphen können also mehrfach durchlaufen werden. Es kann sein, dass Paragraphen gleichzeitig als Teil einer expliziten (`GOTO`- und `PERFORM`-Anweisungen) und einer impliziten Reihenfolge (Fall-Through) durchlaufen werden (siehe Erläuterungen zur Abb. 4). Eine weitere Möglichkeit der Mehrfachbenutzung ist, dass ein und derselbe Paragraph als Teil von verschiedenen expliziten, sich überlappenden Sequenzen von Paragraphen benutzt wird. Für die Analyse (z.B. Program-Slicing) sind solche komplexen Ablaufstrukturen eine sehr interessante Problemstellung.

In *MiCo* gibt es keine Schleifen-Anweisung, die einem `WHILE` oder einem `REPEAT` in anderen prozeduralen Sprachen entsprechen würde. Sie müssen mit Hilfe von Sequenzen von Paragraphen implementiert werden, welche durch `PERFORM` mit den Zusätzen `UNTIL` resp. `TIMES` aufgerufen werden müssen, was das oben angesprochene Problem der Mehrfach-Durchläufe von Programmteilen wiederum verschärft. Zur Veranschaulichung stelle man sich überlappende Sequenzen von Paragraphen vor, die von verschiedenen `PERFORM`-Anweisungen zur Bildung von Schleifen benutzt werden und innerhalb diesen weitere `PERFORM`-Anweisungen, welche ihrerseits wiederum überlappende Sequenzen von Paragraphen aufrufen.

Die Verwendung von `GOTO`-Anweisungen erhöht die Komplexität von Programmen zusätzlich. Nicht nur, dass sie das Verfolgen des Kontrollflusses erschweren, sondern sie können mit `PERFORM`-Anweisungen in fataler Weise interferieren. Bei jedem Aufruf durch ein `PERFORM` wird

Statement	→	MoveStatement $\mid$ IfStatement
	$\mid$	GotoStatement $\mid$ SkipStatement
	$\mid$	PerformStatement
SkipStatement	→	“SKIP“
MoveStatement	→	“MOVE“ AExpression “TO“ Identifier
GotoStatement	→	“GOTO“ Identifier
PerformStatement	→	“PERFORM“ Identifier [“THRU“ Identifier] [UntilClause $\mid$ TimesClause]
UntilClause	→	“UNTIL“ BExpression
TimesClause	→	“TIMES“ AExpression
IfStatement	→	“IF“ BExpression “THEN“ StatementSeq “ELSE“ StatementSeq “END“
BExpression	→	BTerm { “OR“ BTerm }
BTerm	→	BFactor { “AND“ BFactor }
BFactor	→	“NOT“ BFactor $\mid$ “(“ BExpression “)“
	$\mid$	AExpression (“=“ $\mid$ “<“ $\mid$ “>“) AExpression
AExpression	→	ATerm { (“+“ $\mid$ “-“) ATerm }
ATerm	→	AFactor { (“*“ $\mid$ “/“) AFactor }
AFactor	→	Number $\mid$ Identifier
	$\mid$	(“+“ $\mid$ “-“) AFactor $\mid$ “(“ AExpression “)“
Identifier	→	*unspecified*
Number	→	*unspecified*

Abbildung 3: Feinstrukturen von *MiCo*

am Ende des letzten Paragraphen in einer Sequenz die Rücksprungadresse hinterlegt. Beim Erreichen dieser Marke wird direkt hinter das aufrufende `PERFORM` verzweigt resp. zurück gesprungen und anschliessend die Marke gelöscht. Wird nun der Kontrollfluss innerhalb dieser Sequenz von Paragraphen durch eine `GOTO`-Anweisung “ungünstig“ umgebogen, so bleibt die Marke erhalten und kann im Verlaufe des Programms einen unerwünschten “Rücksprung“ provozieren. Ein solches Programm wird sogar vom Programmierer selbst schwer zu verstehen sein.

Wie man solche komplexen Kontrollflüsse mit Hilfe der denotationellen Semantik in den Griff bekommen kann, haben wir in [BFKO93a] gezeigt. Dort wird auch die Standardinterpretation von *MiCo* in denotationeller Semantik definiert.

3.4 Beispiel eines *MiCo*-Programms

Nachdem wir nun die Syntax und einige Eigenheiten vom *MiCo* eingeführt haben, wollen wir uns im nun folgenden Teil ein Beispiel-Programm ansehen und beschreiben. Das *MiCo*-Programm `Factorial-of-6` in Abb. 4 berechnet die Fakultät von 6 (entnommen aus [BFKO93a])

Das Programm ist syntaktisch in eine `DATA DIVISION` und eine `PROCEDURE DIVISION` unterteilt. Letztere ist weiter in drei Paragraphen gegliedert, nämlich `Initialization`, `Control` und `Computation`. Die Paragraphen ihrerseits bestehen aus `MOVE`- und `PERFORM`-Anweisungen. Die `PERFORM`-Anweisung im Paragraphen `Control` implementiert eine While-Schleife mit der Bedingung `(NOT n>2)` und dem Paragraphen `Computation` als Schleifenkörper.

Wie sieht die Ausführung des Programms aus? Zunächst werden ein Zähler `n` und ein Akkumulator `acc` deklariert. Im ersten Paragraphen wird der Zähler `n` mit 6 und der Akkumulator `acc` mit 1 initialisiert. Im folgenden Paragraph `Control` wird mit Hilfe einer `PERFORM`-

```
PROGRAM Factorial-of-6
  DATA DIVISION
    n, acc : NUMERIC
  END
  PROCEDURE DIVISION
    PARAGRAPH Initialization
      MOVE 6 TO n .
      MOVE 1 TO acc
    END
    PARAGRAPH Control
      PERFORM Computation UNTIL NOT n > 2
    END
    PARAGRAPH Computation
      MOVE acc * n TO acc .
      MOVE n - 1 TO n
    END
  END
END
```

Abbildung 4: Ein *MiCo*-Programm zur Berechnung von 6!

Anweisung eine While-Schleife realisiert. Als Schleifenkörper wird der Paragraph `Computation` verwendet, in welchem der Akkumulator `acc` mit dem aktuellen Wert des Zählers `n` multipliziert und anschliessend der Zähler dekrementiert wird. Nachdem die Bedingung der Schleife erfüllt ist, ist das Programm aber noch nicht beendet. Es ist zu berücksichtigen, dass wegen des impliziten Kontrollflusses (Fall-Through) die Kontrolle nach dem Paragraphen `Control` dem Paragraphen `Computation` zurückgegeben wird. Aus diesem Grund muss die Bedingung in der `PERFORM`-Anweisung sicherstellen, dass der Paragraph `Computation` noch einmal durchlaufen werden darf, ohne dass falsche Resultate erzeugt werden.

4 Slicen von *MiCo*-Programmen

4.1 Abstrakter Syntaxbaum

Um das semantik-basierte Program-Slicing von *MiCo* durchführen zu können, muss jede Anweisungen in einem *MiCo*-Programm eindeutig markiert werden können. Um dies zu erreichen muss der Abstrakte Syntax Baum, den wir bereits für die Standardinterpretation von *MiCo* definiert haben [BFKO93a], um den Datentyp `LABEL` erweitert werden (siehe Abb. 5). Jede einzelne Anweisung (und jeder Paragraph) im Originalprogramm erhält eine eindeutige Marke (Nummerierung).

4.2 Semantische Bereiche

In der Abb. 6 sind alle für ein Slicing erforderlichen semantischen Bereiche aufgelistet. Die `LOCATION` bezeichnet eine Speicherstelle und `NUMBER` steht für einen numerischen Wert. Beide werden als ganze Zahlen dargestellt. Der semantische Bereich `VALUE` beschreibt die zulässigen Typen von Werten, die in den Speicher eingetragen werden dürfen. Entweder ist ein Wert noch nicht

```
datatype LABEL         = Label of int | LabelEmpty;
datatype IDENTIFIER    = Identifier of string;
datatype NUMBER        = Number of string;
datatype AEXPRESSION   = (* arithmetic expression *);
datatype BEXPRESSION   = (* boolean expression    *);
datatype PERFORMCLAUSE = PerformClauseUntil of BEXPRESSION
                       | PerformClauseTimes of AEXPRESSION;
datatype PERFORM       = PerformId         of IDENTIFIER
                       | PerformIdId       of IDENTIFIER * IDENTIFIER
                       | PerformIdClause   of IDENTIFIER * PERFORMCLAUSE
                       | PerformIdIdClause of IDENTIFIER * IDENTIFIER * PERFORMCLAUSE;
datatype STATEMENT     = StatementSkip of LABEL
                       | StatementMove of LABEL * (AEXPRESSION * IDENTIFIER)
                       | StatementGoto of LABEL * (IDENTIFIER)
                       | StatementIf   of LABEL * (BEXPRESSION * STATEMENTS * STATEMENTS)
                       | StatementPerform of LABEL * PERFORM
and      STATEMENTS    = StatementSeq of STATEMENT * STATEMENTS | StatementSeqEnd;
datatype PARAGRAPH     = Paragraph of LABEL * (IDENTIFIER * STATEMENTS);
datatype PARAGRAPHS    = ParagraphSeq of PARAGRAPH * PARAGRAPHS | ParagraphSeqEnd;
datatype TYPE          = TypeNumeric;
datatype VARIABLE      = Variable of IDENTIFIER * VARIABLE | VariableEnd;
datatype VARIABLES     = VariableSeq of VARIABLE * VARIABLES | VariableSeqEnd;
datatype DECLARATION   = Declaration of VARIABLES * TYPE;
datatype DECLARATIONS  = DeclarationSeq of DECLARATION * DECLARATIONS | DeclarationSeqEnd;
datatype PROGRAM       = Program of IDENTIFIER * DECLARATIONS * PARAGRAPHS;
```

Abbildung 5: Abstrakte Syntax von *MiCo*

initialisiert (`SvEmpty`) oder eine Continuation (`SvCont`) oder eine Liste von Marken (`SvVar`), welche die Program-Slices zu den Variablen darstellen oder aber ein Tupel (`SvPara`), welches Informationen zu einem Paragraphen beinhaltet. Den Inhalt eines solchen Tupels beschreibt der semantische Bereich `PARA`. Der erste Eintrag bezeichnet eine Funktion von Continuation nach Continuation. Der zweite Eintrag enthält eine Speicheradresse, die besagt, wo im Speicher eine bestimmte Continuation zu finden ist. Der dritte Eintrag schliesslich ist die statisch bestimmte Continuation eines Paragraphen.

Der semantische Bereich `STATE` beschreibt den Zustand des Speichers unmittelbar vor einer Anweisung, die als nächstes ausgewertet werden soll. Für diese Stelle hält er im ersten Eintrag die Wertebelegung als Funktion von Adressen zu Speicherwerten fest. Im zweiten Eintrag werden die Kontrollflussabhängigkeiten für diese Programmstelle mitgeführt.

Der semantische Bereich `ANSWER` repräsentiert die möglichen Resultate einer Analyse. Entweder war sie erfolgreich und der Endzustand (`Answer`) wird zurückgegeben, oder die Analyse war erfolglos (`Error`).

Schliesslich werden im semantischen Bereich `ENVIRONMENT` Angaben über die Umgebung festgehalten. Der erste Eintrag bezeichnet eine Funktion, die jedem Namen im Programm (Variablen oder Paragraphen) eine Adresse im Speicher zuweist. Der zweite Eintrag zeigt auf den nächsten freien Speicherplatz. Der dritte Eintrag ist eine Liste von Tripeln. Zu jedem Paragraphen in einem *MiCo*-Programm existiert genau ein Eintrag in dieser Liste. Jedes Tripel besteht aus dem Paragraphnamen, der Marke des Paragraphen und der Anzahl Anweisungen innerhalb eines Paragraphen. Die Reihenfolge der Tripel in der Liste entspricht der syntaktischen Reihenfolge der

```
type LOCATION   = int;
type NUMBER     = int;

datatype STATE  = State  of ((LOCATION -> VALUE) * LABEL list)
     and ANSWER = Answer of STATE | Error
     and VALUE  = SvEmpty
                | SvCont of (STATE -> ANSWER)
                | SvVar  of LABEL list
                | SvPara of PARA
     and PARA   = Para   of ((STATE -> ANSWER) -> (STATE -> ANSWER))
                          * LOCATION * (STATE -> ANSWER);

datatype ENVIRONMENT = Env of (IDENTIFIER -> LOCATION)
                            * LOCATION
                            * (IDENTIFIER * LABEL * NUMBER) list;
```

Abbildung 6: Semantische Bereiche des Program-Slicers für *MiCo*

Paragraphen im *MiCo*-Programm.

4.3 Hilfsfunktionen

```
subst     : (''a -> 'b) -> ''a -> 'b -> ''a -> 'b
union     : ''a list -> ''a list -> ''a list
#n        : (* Projektion auf die n-te Komponente *)
Y         : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
o         : ('a -> 'b) * ('c -> 'a) -> ('c -> 'b)

isCont    : VALUE -> bool
isPara    : VALUE -> bool
isPerform : STATEMENT -> bool

getCont   : VALUE -> (STATE -> ANSWER)
getPara   : VALUE -> (((STATE->ANSWER) -> (STATE->ANSWER)) * LOCATION * (STATE->ANSWER))
getLab    : (IDENTIFIER * LABEL * 'a) list -> IDENTIFIER -> LABEL
getStm    : (IDENTIFIER * 'a * NUMBER) list -> IDENTIFIER -> NUMBER

aRefs     : AEXPRESSION -> (IDENTIFIER -> VALUE) -> LABEL list
bRefs     : BEXPRESSION -> (IDENTIFIER -> VALUE) -> LABEL list
```

Abbildung 7: Hilfsfunktionen für die semantischen Funktionen

In der Abb. 7 sind die Hilfsfunktionen aufgelistet, welche in den semantischen Funktionen verwendet werden. Die Funktion `subst` substituiert Werte in einer Funktion. Mit `union` werden Mengen vereint, die als Listen repräsentiert werden. Die n-te Komponenten aus einem Tupel kann mit Hilfe der Projektionsfunktion `#n` ermittelt werden. Auch die Fixpunktfunktion `Y` und der Kringel-Operator `o` werden in den semantischen Funktionen benötigt.

Die nächsten Funktionen `isCont`, `isPara` und `isPerform` sind Testfunktionen. Sie kommen dort zur Anwendung, wo bestimmte Voraussetzungen zwingend erfüllt sein müssen (z.B. der betrachtete Wert muss eine Continuation sein).

Die Abfragefunktionen `getCont`, `getPara`, `getLab` und `getStm` werden verwendet, um Werte aus einem Objekt heraus zu schälen.

Mit Hilfe der Funktionen `aRefs` resp. `bRefs` werden in den arithmetischen resp. logischen Ausdrücken die in ihnen enthaltenen Variablen bestimmt und deren zugehörige Slices vereint. Das resultierende Slice (Menge von Marken) enthält alle transitiven Abhängigkeiten der Variablen im bearbeiteten Ausdruck.

4.4 Semantische Funktionen

Obwohl die semantischen Funktionen für das Program-Slicing in der Implementation in Standard ML bottom-up definiert sind (es dürfen nur bereits definierte Funktionen verwendet werden), werden wir sie hier top-down beschreiben, da der Gesamtzusammenhang der Funktionen untereinander so besser gezeigt werden kann.

4.4.1 PRG

```
fun PRG (Program(Identifier id, ds, ps)) e c
    = (fn s => let val ((Env e'), s') = DS ds (e, s)
                   val es = JS ps (Env(#1 e', #2 e', parasInfo ps), s') c
               in (PS ps (#1 es) c) (#2 es) end);
```

Abbildung 8: Die Funktion `PRG`

Die Funktion

`PRG: PROGRAM → ENVIRONMENT → (STATE → ANSWER) → (STATE → ANSWER)`

(siehe Abb. 8) generiert die Slices für alle Variablen des Programms. Dazu verwendet sie den abstrakten Syntaxbaum `Program(Identifier id, ds, ps)`, die initiale Umgebung `e` und die initiale Continuation `c`. Das Resultat ist eine Continuation `(STATE → ANSWER)`, die, angewendet auf einen initialen Anfangszustand `s`, den Endzustand der Berechnung vom Typ `ANSWER` liefert. Dieser Endzustand enthält die Slices für alle Variablen. Zuerst werden durch die Funktion

`DS: DECLARATIONSEQ → ENVIRONMENT → (ENVIRONMENT * STATE)`

die Variablen aus der `DEFINITION DIVISION` des Programms ins Environment übertragen und alle Inhalte der Variablen mit `(SvVar [])`, welches einem leeren Slice entspricht, initialisiert. Die Funktion

`parasInfo: PARAGRAPHSEQ → (IDENTIFIER * LABEL * NUMBER) list`

bildet die Sequenz der Paragraphen des Programms in eine Liste ab und ergänzt jeden Eintrag in der Liste mit der Marke und der Anzahl der Anweisungen des Paragraphen. Diese Angaben werden später bei der Analyse von Schleifen benötigt.

Die Funktion `JS` ergänzt die Umgebung und den aktuellen Zustand mit Angaben aus der `PROCEDURE DIVISION` (siehe Abschnitt 4.4.2). Als letzter Schritt wird sodann die vollständige Continuation des Programms durch die Funktion `PS` generiert, welche angewendet auf den von `JS` aktualisierten Zustand das Resultat vom Typ `ANSWER` liefert (siehe Abschnitt 4.4.3).

4.4.2 JS und J

```
fun JS ParagraphSeqEnd       es _ = es
  | JS (ParagraphSeq(p, ps)) es c
    = let fun F f =
          (fn () =>
              let fun J (Paragraph(_, p')) es' c'
                  = let val (Env e)  = #1 es'
                        val (Env e') = Env (subst (#1 e) (#1 p') (#2 e), (#2 e)+2, #3 e)
                        val c''       = (fn cc => CS (#2 p') (#1 (f())) cc)
                        val (State s)  = #2 es'
                        val (State s') = State (subst (#1 s) (#2 e)
                                              (SvPara(Para(c'', (#2 e)+1, c'))), [])
                    in  (Env e', State(subst (#1 s') ((#2 e)+1) (SvCont c'), (#2 s')))
                    end
                  val c''' = fn s => ((PS ps (#1 (f())) c) s)
              in  JS ps (J p es c''') c end)
      in (Y F) () end;
```

Abbildung 9: Funktionen JS und J

Eine zentrale Rolle in diesem semantik-basierten Ansatz spielt die Vorbereitungsphase durch JS zur eigentlichen Analysephase von PS. Im Gegensatz zum Direct-Style Ansatz von P.A. Hausler und G.A. Venkatesh, wo der Kontrollfluss nicht explizit verfolgt wird, muss in unserem Ansatz jede Continuation einer möglichen Sprungadresse im voraus berechnet und zur Verfügung gestellt werden. Gemeinsam bauen JS und J alle Continuations für alle im Programm verwendbaren Sprungadressen (d.h. für jeden Paragraphen) auf. So können sie bei Bedarf während der eigentlichen Analyse direkt aufgerufen und verwendet werden. Dazu müssen alle im Programm referenzierbaren Sprungadressen, sprich Paragraphnamen, mit ihrer Continuation versehen werden. Das bedeutet, dass im Environment jedem Paragraphnamen eine Location zugeordnet wird und im Speicher des momentanen Zustands die erzeugten Continuations abgelegt werden. Dafür zuständig sind die beiden Funktionen JS und J (siehe Abb. 9). Sie haben die folgenden Signaturen:

JS : PARAGRAPHSEQ → (ENVIRONMENT * STATE) → (STATE → ANSWER) → (ENVIRONMENT * STATE)

J : PARAGRAPH → (ENVIRONMENT * STATE) → (STATE → ANSWER) → (ENVIRONMENT * STATE)

Die Gleichung für eine Sequenz von Paragraphen lautet:

```
JS (p, ps) (e, s) c = JS ps (J p (e, s) c') c
mit c' = PS ps (#1 (JS (p, ps) (e, s) c)) c
```

Die in den Funktionen JS und J verwendeten Funktionen PS und CS werden weiter unten beschrieben. Die Funktion J hat die Aufgabe, den Paragraphnamen ins Environment einzutragen und den momentanen Zustand durch zwei Einträge zu aktualisieren (vergl. Abb. 9). Als erstes wird ein Tripel mit den Angaben zum Paragraphen abgelegt. An der ersten Stelle des Tripels steht eine Funktion mit der Signatur (STATE → ANSWER) → (STATE → ANSWER), welche die Sequenz von Anweisungen im Paragraphen in eine ihr übergebene Continuation integriert. An der zweiten Stelle erscheint die Adresse einer Speicherstelle, wo die dynamisch aktualisierte Continuation zu finden ist, falls der Paragraph durch ein PERFORM aufgerufen wurde. An der dritten Stelle schliesslich wird die statische Continuation festgehalten, welche dann Verwendung findet, wenn nach Beenden des Paragraphen sein syntaktischer Nachfolger (Fall-Through) bearbeitet werden soll.

Es ist darauf hinzuweisen, dass die Funktionen `PS` und `CS`, welche innerhalb der Funktionen `JS` resp. `J` aufgerufen werden, auf ein Environment zugreifen, wie es eigentlich erst nach Beendigung der Funktionen `JS` resp. `J` in Erscheinung treten kann!

4.4.3 PS und P

Die Funktion `PS` in Abb. 10 generiert die Continuation für eine Sequenz von Paragraphen. `PS` gibt die Continuation `c` unverändert zurück, falls das Ende der Sequenz erreicht wurde, andernfalls ruft sie die Funktion `P` auf. Dabei wird die Continuation `c` nicht weitergereicht, da die Funktion `P` ihre Continuation anhand des Paragraphnamens und des übergebenen Environments `e` selber bestimmt.

```
fun PS ParagraphSeqEnd      _ c = c
  | PS (ParagraphSeq(p, _)) e _ = P p e;

fun P (Paragraph(_, (id, _))) (Env e)
    = (fn (State s) =>
           let val p' = ((#1 s) o (#1 e)) id
           in if isPara(p')
              then let val p = getPara(p')
                   in  if isCont((#1 s) (#2 p))
                        then ((#1 p) (getCont((#1 s) (#2 p))))
                             (State (subst (#1 s) (#2 p) (SvCont (#3 p)), #2 s))
                        else Error end
              else Error end);
```

Abbildung 10: Funktionen `PS` und `P`

Wie man in Abb. 10 sieht, vernachlässigt die Funktion `P` die Sequenz der Anweisungen im Paragraphen, denn anhand des Paragraphnamen wird aus des ersten Eintrags im Environment das entsprechende Tripel mit den Informationen zum Paragraphen aus dem aktuellen Zustand gelesen. Erinnern wir uns, der erste Eintrag in diesem Tripel ist eine Funktion, welche die Sequenz der Anweisungen im Paragraphen in die ihr übergebene Continuation einarbeitet. Diese Funktion wurde bereits in der Vorbereitungsphase durch `JS` resp. `J` berechnet und bereitgestellt. Mit anderen Worten, durch Anwendung dieser Funktion aus dem Tripel sind die Anweisungen im Paragraphen bereits bearbeitet. Als Argument wird dieser Funktion die dynamische Continuation aus dem aktuellen Zustand übergeben. Die resultierende Continuation wird dann auf den angepassten Zustand angewandt, wobei die Anpassung darin besteht, dass die eben verwendete dynamische Continuation auf die statische Continuation zurückgesetzt wird.

4.4.4 CS

Die Funktion `CS` in Abb. 11 berechnet die Continuation einer Sequenz von Anweisungen. In einem ersten Schritt wird die Continuation der Sequenz `sts` berechnet. Sie wird anschliessend von der Funktion `C` (siehe Abb. 12 bis 18) verwendet, um die Effekte der ersten Anweisung `st`

```
fun CS StatementSeqEnd         _ c = c
  | CS (StatementSeq(st, sts)) (Env e) c
    = C st (Env e) (
      if isPerform(st)
      then (fn (State s) =>
            let val p' = ((#1 s) o (#1 e)) (lastPara(st))
            in  if isPara(p')
                then let val p = getPara(p')
                     in if isCont((#1 s) (#2 p))
                        then (CS sts (Env e) c)
                             (State (subst (#1 s) (#2 p) (SvCont(#3 p)),(#2 s)))
                        else Error end
                else Error end)
      else CS sts (Env e) c);
```

Abbildung 11: Funktion CS

darin zu integrieren. Bei der Berechnung der Effekte von sts müssen PERFORM-Anweisungen speziell behandelt werden. Warum das? Ein PERFORM hat die Wirkung eines Prozeduraufrufs, d.h., dass nach Ausführung des Paragraphen (oder Sequenz von Paragraphen) an die Stelle direkt nach dem Aufruf zurückgesprungen wird. Alle Modifikationen der dynamischen Continuation (siehe Abb. 17 und 18) eines angesprungenen Paragraphen müssen wieder rückgängig gemacht werden, indem der betreffende Speicherplatz im aktuellen Zustand mit der statischen Continuation überschrieben wird. Diese Modifikationen betreffen immer den letzten Paragraphen in einer Sequenz, welcher durch die Funktion

lastPara : STATEMENT → IDENTIFIER

bestimmt wird.

4.4.5 C

- **SKIP**

```
C (StatementSkip l) _ c =  c
```

Abbildung 12: Funktion C für die Anweisung SKIP

Die SKIP-Anweisung hat keine Effekte, weder auf das Environment, noch auf die Continuation oder den momentanen Zustand. Darum wird die übergebene Continuation unverändert zurückgeliefert.

- **MOVE**

Die MOVE-Anweisung entspricht einer Zuweisung. Sie hat für den Program-Slicer eine ausserordentliche Wichtigkeit, da sie die Datenabhängigkeit zwischen der neu zu berechnenden Variablen id und den verwendeten Variablen im arithmetischen Ausdruck a festhält. Der neue Slice für die Variable id wird konstruiert aus der Marke der MOVE-Anweisung und aus den zu den Variablen im arithmetischen Ausdruck a zugehörigen Slices. Das Sammeln der Marken aus a geschieht durch die Funktion aRefs (siehe Abb. 7). Die transitiven

```
C (StatementMove(l, (a, id))) (Env e) c
= (fn (State s) =>
      let val sv = SvVar(union(union [l] (aRefs a ((#1 s) o (#1 e)))) (#2 s))
      in c (State (subst (#1 s) ((#1 e) id) sv, (#2 s))) end)
```

Abbildung 13: Funktion `C` für die Anweisung `MOVE`

Abhängigkeiten von `id` zu weiteren Anweisungen werden von `aRefs` automatisch erfasst (siehe Abschnitt 4.3). Falls eine Zuweisung innerhalb einer bedingten Anweisung oder einer Schleife Verwendung findet, müssen diese für das Konstruieren des Program-Slices der Variablen `id` berücksichtigt werden. Aus diesem Grund wird der zweite Eintrag im Zustand `s`, welche die Kontrollabhängigkeiten für diesen Programpunkt festhält, ebenfalls in das Program-Slice von `id` aufgenommen.

- **IF**

```
C (StatementIf(_, (b, sts1, sts2))) (Env e) c
= (fn (State s) =>
      let val s'  = ((#1 s), union (bRefs b ((#1 s) o (#1 e))) (#2 s))
          val c'  = (fn (State ss) => c (State ((#1 ss), (#2 s))))
          val s'' = lub (Env e) ((CS sts1 (Env e) c') (State s'))
                                ((CS sts2 (Env e) c') (State s'))
      in resetContAndDep (Env e) s'' (State s) end)
```

Abbildung 14: Funktion `C` für die Anweisung `IF...THEN...ELSE`

Bei einer bedingten Anweisung wird in der Standardinterpretation einer der beiden Pfade ausgewählt und durchlaufen. Beim vorliegenden statischen Slicing kann die Bedingung nicht ausgewertet werden, da das Programm nicht ausgeführt wird. Es fehlt also die Information, welcher der Pfade schlussendlich durchlaufen wird. Da aber beide potentiell durchlaufen werden können, müssen auch beide Pfade in die statische Analyse einbezogen werden. Anschliessend werden die Resultate beider Äste zusammengefasst, um die Effekte beider Äste in den Slice zu übernehmen.

Als Erstes werden mit Hilfe der Funktion `bRefs` alle transitiven Abhängigkeiten der Variablen in der Bedingung gesammelt (siehe Abb. 7). Alle Anweisungen in den beiden Ästen stehen in einer Kontrollabhängigkeit zu den Variablen in der Bedingung. Sie werden gemeinsam mit schon vorhanden Kontrollabhängigkeiten in einen neuen Zustand übertragen, der unmittelbar nach der Bedingung, aber noch vor allen Anweisungen in beiden Pfaden gültig ist.

Die Kontrollabhängigkeiten von den Variablen in der Bedingung gelten nur innerhalb der beiden Äste. Nach der bedingten Anweisung müssen die vorher gültigen Kontrollabhängigkeiten wieder hergestellt werden. Dies wird dadurch erreicht, dass die Continuation `c'` dahingehend modifiziert wird.

Die beiden Sequenzen von Anweisungen `sts1` und `sts2` werden unabhängig voneinander analysiert. Jeder Ast wird separat bis zum Ende des Programms analysiert. Anschliessend werden die beiden Resultate durch die Funktion

```
lub : ENVIRONMENT → ANSWER → ANSWER → ANSWER
```

(least upper bound) zu einem Resultat zusammen geführt. Konkret heisst das, dass die Abhängigkeiten einer Variablen beim Durchlaufen eines Astes mit den Abhängigkeiten der gleichen Variablen beim Durchlaufen des anderen Astes vereint werden. Die Funktion

```
resetContAndDep : ENVIRONMENT → ANSWER → STATE → ANSWER
```

stellt für die Kontrollabhängigkeiten und Continuations ihre ursprünglichen Werte vor der bedingten Anweisung wieder her.

- **GOTO**

```
C (StatementGoto(l, id)) (Env e) _
= (fn (State s) =>
       let val p' = ((#1 s) o (#1 e)) id
       in  if isPara(p')
           then let val p = getPara(p')
                in if isCont((#1 s) (#2 p))
                   then let val s' = updateState (Env e) (State s)
                                     (union [l, (getLab (#3 e) id)] (#2 s))
                        in ((#1 p) (getCont((#1 s) (#2 p)))) s' end
                   else Error end
           else Error end)
```

Abbildung 15: Funktion `C` für die Anweisung `GOTO`

Die `GOTO`-Anweisung ist ein unbedingter Sprungbefehl, d.h. der Kontrollfluss wird ohne weitere Vorbedingung zu einen anderen Programmpunkt umgelenkt. Dies erklärt auch, warum die übergebene Continuation vernachlässigt wird. Die eigentliche Continuation wird durch die `GOTO`-Anweisung selbst berechnet. In der vorliegenden Implementation werden nur die vorwärts orientierten Sprünge korrekt bearbeitet. Die rückwärts orientierten Sprünge bewirkt, dass die Analyse nicht terminiert.

Die `GOTO`-Anweisung kann auf zwei unterschiedliche Arten verstanden werden:

- Ihre eigentliche Aufgabe ist, den Kontrollfluss dorthin lenkt, wo die Berechnung weitergeführt werden soll. Alle nach einer solchen `GOTO`-Anweisung im Programm verwendeten Variablen werden durch diesen Programmpunkt beeinflusst, da jeder Pfad zu ihnen zwangsläufig über diese `GOTO`-Anweisung führt.
- Weiter kann die `GOTO`-Anweisung so aufgefasst werden, dass sie bestimmte Programmteile durch überspringen für das Fall-Through (syntaktische Reihenfolge der Paragraphen) ausblendet. Ein solches Beispiel stellt das *MiCo*-Programm in Abb. 16 dar.

  Hier stellt die `GOTO`-Anweisung sicher, dass der Paragraph `WhileLoop` ausschliesslich von der `PERFORM`-Anweisung in der Zeile 4 erreicht werden kann. Der Paragraph `WhileLoop` wird nie von der Regel des Fall-Through erfasst werden. Das bedeutet,

```
PROGRAM GotoTest .
  DATA DIVISION .
    x, y : NUMERIC
  END
  PROCEDURE DIVISION .
    PARAGRAPH Initialization .              (*>1<*)
      MOVE 0 TO y .                         (*>2<*)
      MOVE 3 TO x .                         (*>3<*)
      PERFORM WhileLoop UNTIL NOT y > 11 .  (*>4<*)
      GOTO Exit                             (*>5<*)
    END .
    PARAGRAPH WhileLoop .                   (*>6<*)
      MOVE x + y TO y                       (*>7<*)
    END .
    PARAGRAPH Exit .                        (*>8<*)
      SKIP                                  (*>9<*)
    END
  END
END
```

Abbildung 16: Programmbeispiel zu `GOTO`

dass die `GOTO`-Anweisung auch gewisse Variablen beeinflusst, konkret die im übersprungenen Paragraphen neu definierten, und darum auch in deren jeweiligen Slices zu integrieren ist.

Die `GOTO`-Anweisung hat demzufolge Wirkungen sowohl auf noch zu analysierende als auch auf bereits analysierte Anweisungen. Um auf der sicheren Seite zu sein, wird diese Stelle im Programm und die momentan geltenden anderen Kontrollabhängigkeiten in alle Slices im aktuellen Zustand `s'` eingebaut. Somit sind die Effekte einer `GOTO`-Anweisung in allen bereits berechneten Slices integriert. Es kann aber vorkommen, dass eine `GOTO`-Anweisung in einen Slice integriert wird, bei dem es nicht notwendig wäre. Im Beispiel 16 ist dies für das Slice der Variablen `x` der Fall. Dies kommt daher, dass `x` zwar *vor* der `GOTO`-Anweisung analysiert werde, aber nicht im übersprungenen Paragraphen neu definiert wird.

Alle Kontrollabhängigkeiten sind nun in die Slices integriert, so dass diese für die Fortsetzung der Analyse mit der leeren Liste initialisiert werden können. Diese entsprechenden Anpassungen werden durch die Funktion

```
updateState : ENVIRONMENT → STATE → LABEL list → STATE
```

durchgeführt, welche im dritten Argument die angesprochenen Abhängigkeiten entgegen nimmt und den neuen modifizierten Zustand zurück liefert.

Die Sprungadresse in der `GOTO`-Anweisung bezeichnet einen Paragraphen. In dessen zugehörigen Tripel von Type `PARA` finden sich alle Daten, die zur Bildung der neuen Continuation benötigt werden. Der erste Eintrag ist eine Funktion, die einer ihr übergebenen Continuation die Anweisungssequenz des Paragraphen einarbeitet; der zweite Eintrag ist eine Speicheradresse, an welcher die dynamische Continuation dieses Paragraphen zu finden ist. Der dritte Eintrag, welcher allerdings hier nicht Verwendung findet, hält die statische Continuation des Paragraphen fest.

- **PERFORM**

```
C (StatementPerform(l, PerformId id)) (Env e) c
= (fn (State s) =>
       let val p' = ((#1 s) o (#1 e)) id
       in  if isPara(p')
           then let val p = getPara(p')
                in if isCont((#1 s) (#2 p))
                   then let val c' = (fn (State ss) =>
                                      c (State ((#1 ss), (#2 s))))
                            val s' = (subst (#1 s) (#2 p) (SvCont c'),
                                      union [l] (#2 s))
                        in ((#1 p) c') (State s') end
                   else Error end
           else Error end)
```

Abbildung 17: Funktion `C` für die Anweisung `PERFORM`

Die `PERFORM`-Anweisung ist ein Subroutinenaufruf, welcher den Kontrollfluss beeinflusst. Nach Beendigung der Subroutine kehrt der Kontrollfluss, im Gegensatz zur `GOTO`-Anweisung, direkt hinter den Ausgangspunkt des Sprungs zurück. Das hat zur Konsequenz, dass die Continuation `c'` der `PERFORM`-Anweisung dahin gehend anzupassen ist, dass die Kontrollabhängigkeiten im zweiten Eintrag des Zustandes jeweils auf den Wert vor der Anweisung zurück gesetzt wird.

Der Zustand `s'`, welcher beim Bearbeiten der Subroutine verwendet wird, erfährt die folgenden zwei Anpassungen. Die erste betrifft die dynamische Continuation des aufzurufenden Paragraphen. Diese wird auf die Continuation `c'` gesetzt (also auf dieselbe, die für die `PERFORM`-Anweisung gilt), um sicher zu stellen, dass nach der Bearbeitung des Paragraphen nicht der syntaktische Nachfolger des Paragraphen ("Fall-Through"), sondern die nächste Anweisung unmittelbar nach der `PERFORM`-Anweisung auszuführen ist. Die zweite Anpassung betrifft die Kontrollabhängigkeiten (zweiter Eintrag im aktuellen Zustand). Alle Anweisungen im aufgerufenen Paragraphen stehen nämlich in einer Kontrollabhängigkeit von dieser `PERFORM`-Anweisung; also muss die Marke dieser `PERFORM`-Anweisung in die Kontrollabhängigkeit eingeführt werden.

- **PERFORM...UNTIL**

  Diese Anweisung ermöglicht die Implementierung einer While-Schleife. Da der Kontrollfluss am Ende des Schleifenkörpers wieder an den Anfang des Schleifenkörpers zurück kehrt, hat jede Anweisung innerhalb der Schleife alle anderen Anweisungen in der Schleife auch als Vorgänger. In der Standardinterpretation wird die Schleife so oft durchlaufen, bis die Schleifenbedingung (z.B. `(x>0) AND (y<2)`) zutrifft. Dazu verwendet man dort eine Fixpunkt-Funktion, die genau dann terminiert, wenn die Bedingung der Schleife erfüllt ist. Es kann also durchaus vorkommen, dass die Standardinterpretation, je nach Schleifenbedingung, nicht terminiert. Bei unserem vorgeschlagenen Program-Slicer sieht dies anders aus:

```
C (StatementPerform(l, PerformIdClause(id, PerformClauseUntil b))) (Env e) c
= (fn (State s) =>
       let val n  = getStm (#3 e) id
           val c' = (fn (State ss) => c (State ((#1 ss), (#2 s))))
           fun F g i = (fn (State s'') =>
               if i>0 then
               let val s' = State((#1 s''),
                                  union (bRefs b ((#1 s'') o (#1 e))) (#2 s''))
               in  lub (Env e) ((C_slc (AstStatementPerform(l,AstPerformId id))
                                (Env e) (g (i-1)) s')) (c' s') end
               else c' (State s''))
         in ((Y F) n (State s)) end)
```

Abbildung 18: Funktion C für die Anweisung PERFORM...UNTIL

1. Die Inhalte der Variablen sind abstrakte Werte. In unserer Interpretation sind dies die zu den Variabeln zugehörigen Program-Slices.
2. In der von uns verwendeten statischen Analyse werden die syntaktischen Elemente nicht unter Verwendung der konkreten Operationen verknüpft, sondern abstrakt interpretiert. Es sind nicht die numerischen resp. boolschen Werte der Variablen entscheidend, sondern die Beziehungen und Abhängigkeiten untereinander.

Da bei einer statischen Interpretation nicht entschieden werden kann, wie oft eine Schleife während der Programmausführung durchlaufen wird, müssen alle Fälle abgedeckt werden. Daraus ergibt sich, dass bei der statischen Analyse einer Schleife der Schleifenkörper so oft durchlaufen werden muss, bis alle Beziehungen und Abhängigkeiten erfasst wurden. Ein solches Abbruchkriterium ist unabhängig von der dynamischen Auswertung der Schleifenbedingung. In [Hau89] wird ein Abbruchkriterium vorgeschlagen und formal bewiesen, welches auf der Anzahl der Anweisungen des Schleifenkörpers (bzw. des Paragraphen) basiert. Nach [Hau89] muss eine Schleife, um alle Abhängigkeiten unter den Variablen zu erfassen, genau so oft analysiert werden, wie sie Anweisungen enthält. Dieses Abbruchkriterium kommt auch in unserem Ansatz zum Einsatz. Dadurch, dass das Abbruchkriterium nur von der Anzahl der Anweisungen innerhalb eines Schleifenkörpers abhängt, terminiert das Program-Slicing immer. Das bedeutet, dass auch ein Programm, das eine Endlos-Schleife enthält, analysiert werden kann.

In der Vorbereitungsphase wurden bereits Informationen zu den einzelnen Paragraphen gesammelt; u.a. auch die Anzahl der Anweisungen. Diese wird mit Hilfe der Funktion `getStm` (siehe Abb. 7), welcher der dritte Eintrag des Environments `e` und der Paragraphnamen `id` übergeben wird, ermittelt. Der Wert wird an `n` gebunden, welcher die Anzahl der Iterationen der Analyse beschränkt (vergl. Abb. 18).

Mit den zu Abb. 17 und 18 gegebenen Beschreibungen lassen sich die weiteren Variationen der PERFORM-Anweisung ableiten. Diese sind PERFORM...THRU, PERFORM...THRU...UNTIL, PERFORM...TIMES und PERFORM...THRU...TIMES.

4.5 Beispiel zum Program-Slicing

```
PROGRAM Example .
  DATA DIVISION .
    i, k, v, x, y, z : NUMERIC
  END
  PROCEDURE DIVISION .
    PARAGRAPH Initialization .          (*>1<*)
      MOVE 0 TO i .                     (*>2<*)
      MOVE 0 TO v .                     (*>3<*)
      MOVE 1 TO x .                     (*>4<*)
      MOVE 2 TO y .                     (*>5<*)
      MOVE 3 TO z .                     (*>6<*)
      PERFORM WhileLoop UNTIL i > 10 . (*>7<*)
      MOVE v TO k .                     (*>8<*)
      GOTO Exit                         (*>9<*)
    END .
    PARAGRAPH WhileLoop .              (*>10<*)
      MOVE z TO v .                    (*>11<*)
      MOVE y TO z .                    (*>12<*)
      MOVE x TO y .                    (*>13<*)
      MOVE i + 1 TO i                  (*>14<*)
    END .
    PARAGRAPH Exit .                   (*>15<*)
      SKIP                             (*>16<*)
    END
  END
END
```

Abbildung 19: Das *MiCo*-Programm `Example`

Nachdem nun die semantischen Funktionen definiert worden sind, wollen wir an einem Beispiel sehen, wie diese die Slices erzeugen. Dabei werden wir Schritt für Schritt die Analyse verfolgen und den jeweils resultierenden Zustand betrachten. Der vollständige Ablauf der Analyse anhand dieses Beispiels findet sich im Anhang A.

1. Initialisieren der Inhalte der Variablen und der Kontrollabhängigkeiten:

   $s_0 = ((\{i \to \{\}, v \to \{\}, k \to \{\}, x \to \{\}, y \to \{\}, z \to \{\}), \{\})$

2. Bei der Analyse einer `MOVE`-Anweisung wird die neu zu berechnende Variable aktualisiert (vergl. Abb. 13). Der neue Slice dieser Variable besteht aus der Marke der `MOVE`-Anweisung, der Vereinigung aller Slices der Variablen im arithmetischen Ausdruck und schliesslich noch aus den an diesem Programmpunkt geltenden Kontrollabhängigkeiten. Bei der ersten Anweisung `MOVE 0 TO i` wird demanch `i` neu definiert. Die Marke dieser Anweisung ist 2. Der arithmetische Ausdruck besteht aus einer Konstanten, weshalb von dieser Seite kein Beitrag zum Slice beigesteuert wird. Kontrollabhängigkeiten existieren momentan ebenfalls noch keine. Somit ergibt sich nach der `MOVE`-Anweisung ein neuer Zustand, der wie folgt aussieht:

   $s_1 = ((\{i \to \{2\}, v \to \{\}, k \to \{\}, x \to \{\}, y \to \{\}, z \to \{\}), \{\})$

Die folgenden `MOVE`-Anweisungen werden in gleicher Weise bearbeitet, wobei die Analyse die nachstehende Sequenz von Zuständen liefert:

$s_2 = ((\{i \to \{2\}, v \to \{3\}, k \to \{\}, x \to \{\}, y \to \{\}, z \to \{\}), \{\})$
$s_3 = ((\{i \to \{2\}, v \to \{3\}, k \to \{\}, x \to \{4\}, y \to \{\}, z \to \{\}), \{\})$
$s_4 = ((\{i \to \{2\}, v \to \{3\}, k \to \{\}, x \to \{4\}, y \to \{5\}, z \to \{\}), \{\})$
$s_5 = ((\{i \to \{2\}, v \to \{3\}, k \to \{\}, x \to \{4\}, y \to \{5\}, z \to \{6\}), \{\})$

3. Die nächste zu analysierende Anweisung ist eine `PERFORM`-Anweisung. Sie verwendet den Paragraphen `WhileLoop`, um eine Schleife zu implementieren. Wie aus Abb. 18 abzuleiten ist, muss der Scheifenkörper einer Schleife höchstens so oft analysiert werden, wie er Anweisungen enthält. Im vorliegenden Fall ist also der Schleifenkörper maximal viermal zu analysieren.

   Der Grund dafür ist darin zu suchen, dass wir in der statischen Analyse nicht wissen, wie oft der Schleifenkörper während des Programmablaufs wirklich durchlaufen wird. Wäre z.B. bei der Initialisierung der Wert der Variablen `i` auf `10` statt auf `0` gesetzt worden, so würde die Schleife nie durchlaufen werden und die Variable `k` würde nur von Anweisungen ausserhalb der Schleife abhängen.

   Damit wir einen Program-Slice erhalten, welcher *alle* Wertebelegungen der Variablen abdeckt, muss auch *jede* Anzahl Durchläufe, die unterschiedliche Slices generieren können, berücksichtigt werden. Im vorliegenden Beispiel heisst das konkret, dass der Slices für eine Variable das null-, ein-, zwei-, drei- oder viermalige Durchlaufen des Schleifenkörpers abdecken muss. Die einzelnen Fälle werden separat berechnet und anschliessend zu einem Program-Slice zusammen geführt.

   Nullmal durchlaufen. Der Schleifenkörper wird ausgelassen:
   Die `MOVE`-Anweisung, die unmittelbar an die `PERFORM`-Anweisung anschliesst, erzeugt den folgenden Zustand:
   $s_6 = ((\{i \to \{2\}, v \to \{3\}, k \to \{3, 8\}, x \to \{4\}, y \to \{5\}, z \to \{6\}), \{\})$
   Bei einer `GOTO`-Anweisung werden *alle* Slices ergänzt mit der Marke der `GOTO`-Anweisung und der Marke des Paragraphen, wohin verzweigt wird:
   $s_7 = ((\{i \to \{2, 9, 15\}, v \to \{3, 9, 15\}, k \to \{3, 8, 9, 15\},$
   $x \to \{4, 9, 15\}, y \to \{5, 9, 15\}, z \to \{6, 9, 15\}), \{\})$
   Aus der `SKIP`-Anweisung im Paragraphen `Exit` resultieren keine Beiträge an die Program-Slices der Variablen:
   $s_8 = ((\{i \to \{2, 9, 15\}, v \to \{3, 9, 15\}, k \to \{3, 8, 9, 15\},$
   $x \to \{4, 9, 15\}, y \to \{5, 9, 15\}, z \to \{6, 9, 15\}), \{\})$

   Einmal durchlaufen. Ausgangszustand für die nachfolgenden Betrachungen ist Zustand s_5, der unmittelbar vor der PERFORM-Anweisung gilt:
   $s_9 = ((\{i \to \{2\}, v \to \{3\}, k \to \{\},$
   $x \to \{4\}, y \to \{5\}, z \to \{6\}), \{2, 7\})$
   $\vdots$
   $s_{17} = ((\{i \to \{2, 7, 9, 14, 15\}, v \to \{2, 6, 7, 9, 11, 15\}, k \to \{2, 6, 7, 8, 9, 11, 15\},$
   $x \to \{4, 9, 15\}, y \to \{2, 4, 7, 9, 13, 15\}, z \to \{2, 5, 7, 9, 12, 15\}), \{\})$
   Die Kontrollabhängigkeiten sind nach dem Analysieren des Schleifenkörpers wieder

in ihren ursprünglichen Zustand zurückgesetzt worden. Die Anweisungen unmittelbar nach der `PERFORM`-Anweisung (`MOVE`..., `GOTO`... und `SKIP`...) sind ebenfalls analysiert und ihre Effekte bereits im Zustand s_{17} integriert.

Zweimal durchlaufen. Ausgangszustand für den zweiten Durchlauf der Analyse ist der Zustand s_{14} (Anhang A), der unmittelbar nach dem Rücksetzen der Kontrollabhängigkeiten gilt:

$$\begin{aligned}
s_{18} = ((\{&i \rightarrow \{2,7,14\}, v \rightarrow \{2,6,7,11\}, k \rightarrow \{\},\\
&x \rightarrow \{4\}, y \rightarrow \{2,4,7,13\}, z \rightarrow \{2,5,7,12\}), \{2,7,14\})\\
\vdots&\\
s_{26} = ((\{&i \rightarrow \{2,7,9,14,15\}, v \rightarrow \{2,5,7,9,11,12,14,15\},\\
&k \rightarrow \{2,5,7,8,9,11,12,14,15\}, x \rightarrow \{4,9,15\},\\
&y \rightarrow \{2,4,7,9,13,14,15\}, z \rightarrow \{2,4,7,9,12,13,14,15\}), \{\})
\end{aligned}$$

Dreimal durchlaufen. Ausgangszustand für den dritten Durchgang der Analyse ist der Zustand s_{23} (Anhang A), der unmittelbar nach dem Rücksetzen der Kontrollabhängigkeiten gilt:

$$\begin{aligned}
s_{27} = ((\{&i \rightarrow \{2,7,14\}, v \rightarrow \{2,5,7,11,12,14\},\\
&k \rightarrow \{\}, x \rightarrow \{4\},\\
&y \rightarrow \{2,4,7,13,14\}, z \rightarrow \{2,4,7,12,13,14\}), \{2,7,14\})\\
\vdots&\\
s_{35} = ((\{&i \rightarrow \{2,7,9,14,15\}, v \rightarrow \{2,4,7,9,11,12,13,14,15\},\\
&k \rightarrow \{2,4,7,8,9,11,12,13,14,15\}, x \rightarrow \{4,9,15\},\\
&y \rightarrow \{2,4,7,9,13,14,15\}, z \rightarrow \{2,4,7,9,12,13,14,15\}), \{\})
\end{aligned}$$

Viermal durchlaufen. Ausgangszustand für den vierten Durchlauf der Analyse ist der Zustand s_{32} (Anhang A), der unmittelbar nach dem Rücksetzen der Kontrollabhängigkeiten gilt:

$$\begin{aligned}
s_{36} = ((\{&i \rightarrow \{2,7,14\}, v \rightarrow \{2,4,7,11,12,13,14\},\\
&k \rightarrow \{\}, x \rightarrow \{4\},\\
&y \rightarrow \{2,4,7,13,14\}, z \rightarrow \{2,4,7,12,13,14\}), \{2,7,14\})\\
\vdots&\\
s_{44} = ((\{&i \rightarrow \{2,7,9,14,15\}, v \rightarrow \{2,4,7,9,11,12,13,14,15\},\\
&k \rightarrow \{2,4,7,8,9,11,12,13,14,15\}, x \rightarrow \{4,9,15\},\\
&y \rightarrow \{2,4,7,9,13,14,15\}, z \rightarrow \{2,4,7,9,12,13,14,15\}), \{\})
\end{aligned}$$

Am Ende der Analyse werden nun die Resultate der verschiedenen Durchläufe zusammen geführt. Endresultat der jeweiligen Analysen sind die Zustände s_8, s_{17}, s_{26}, s_{35} und s_{44}. Damit die Slices des Programmes *alle* Fälle abdecken, müssen diese Zustände nun kombiniert werden. Dabei werden die Inhalte (Mengen von Marken) der Variablen mit gleichem Namen aus den Zuständen miteinander vereint. Der Endzustand der Analyse präsentiert sich wie folgt:

$$
\begin{aligned}
&\vdots \\
s_{48} = ((\{&i \to \{2,7,9,14,15\}, v \to \{2,3,4,5,6,7,9,11,12,13,14,15\}, \\
&k \to \{2,3,4,5,6,7,8,9,11,12,13,14,15\}, x \to \{4,9,15\}, \\
&y \to \{2,4,5,7,9,13,14,15\}, z \to \{2,4,5,6,7,9,12,13,14,15\}), \{\})
\end{aligned}
$$

In diesem Endzustand s_{48} sind zu jeder Variablen ihr berechneter Program-Slice festgehalten. Die Slices werden als Mengen von Marken dargestellt. Diese bezeichnen genau diejenigen Anweisungen aus dem Originalprogramm, welche auf eine zu untersuchende Variable Einfluss nehmen können.

5 Zeitkomplexität des Ansatzes

5.1 Motivation

Im vorangegangenen Abschnitt haben wir den Program-Slicer mit Hilfe der denotationellen Semantik beschrieben. In diesem Abschnitt soll nun der Aufwand für das Berechnen der Program-Slices untersucht werden. Es ist vorauszuschicken, dass die vorgeschlagene Masszahl nicht dazu dient, die Komplexität eines Programms zu bewerten. Sie wird vielmehr dazu verwendet, den Aufwand des Ansatzes zur Erzeugung *aller* Slices eines Programms zu messen. Das heisst, dass die Masszahl unabhängig von der Länge der einzelnen Slices eines Programms sein wird, weil der Algorithmus alle Program-Slices gleichzeitig berechnet.

In diesem Abschnitt wird die Masszahl mz für den Berechnungsaufwand unseres Ansatzes eingeführt. Zunächst werden wir an einem Beispiel zeigen, wie sich der Aufwand zur Analyse eines *MiCo*-Programms bestimmen lässt. Die Anweisungen werden dabei unterschiedlich gewichtet. Eine Begründung für die gewählte Gewichtung wird ebenfalls gegeben. Im weiteren soll dann die Berechnung der Masszahl mz basierend auf der denotationellen Semantik beschrieben werden.

5.2 Herleiten des Berechnungsaufwandes am Beispiel

Die Gewichte der Anweisungen aus *MiCo* sollen den Werten, wie sie in der Abb. 20 beschrieben sind, entsprechen. Wie lässt sich nun die Masszahl mz (Aufwand zur Berechnung der Program-Slices) für das *MiCo*-Programm im Beispiel in Abb. 21 bestimmen?

`SKIP`	0
`MOVE`	1
`GOTO`	1
`IF-THEN-ELSE`	2
`PERFORM`	3

Abbildung 20: Gewichtung der *MiCo*-Anweisungen für das Program-Slicing

Bei der Analyse wird der Paragraph `Initialization` einmal durchlaufen. Die in ihm enthaltenen beiden Zuweisungen erhöhen den Wert von mz auf 2. Der Kontrollfluss wechselt dann in den Paragraphen `Control`. Dieser besteht nur aus einem `PERFORM...UNTIL`, welches eine Schleife implementiert mit dem Paragraphen `Computation` als Schleifenkörper. Unser Ansatz schreibt

```
PROGRAM Factorial-of-6
  DATA DIVISION
    n, acc : NUMERIC
  END
  PROCEDURE DIVISION
    PARAGRAPH Initialization
      MOVE 6 TO n .
      MOVE 1 TO acc
    END
    PARAGRAPH Control
      PERFORM Computation UNTIL NOT n > 2
    END
    PARAGRAPH Computation
      MOVE acc * n TO acc .
      MOVE n - 1 TO n
    END
  END
END
```

Abbildung 21: Ein *MiCo*-Programm mit Berechnungsaufwand $mz = 18$

vor, dass bei der Analyse einer Schleife, diese höchstens so oft durchlaufen wird, als sie Zuweisungen enthält. Im vorliegenden Beispiel enthält der Schleifenkörper 2 Zuweisungen, weshalb er nicht mehr als zweimal durchlaufen werden muss.

Da bei einer statischen Analyse nicht bekannt ist, wie oft die Schleife in einer dynamischen Interpretation ausgeführt wird, müssen alle Fälle betrachtet werden, die unterschiedliche Resultate liefern können. Im Programm `Factorial-of-6` gibt es diesbezüglich drei Fälle zu berücksichtigen.

1. Der Schleifenkörper muss nullmal analysiert werden, für den Fall, dass die Variable `n` mit einer Zahl kleiner oder gleich 2 initialisiert wurde.

2. Für den Fall, dass `n` mit 3 initialisiert wurde, muss der Schleifenkörper genau einmal analysiert werden.

3. In allen anderen Fällen wird der Schleifenkörper zweimal analysiert werden.

Im ersten Fall wird die Schleife übergangen und der Paragraph `Computation` analysiert (Fall-Through). Die beiden Zuweisungen in diesem Paragraphen erhöhen den Betrag des Aufwandes um 2 auf 4.

Im zweiten Fall wird Schleife wird das `PERFORM` einmal analysiert. Die `PERFORM`-Anweisung erhöht den Wert von mz um den Betrag 3 und die beiden Anweisungen im Schleifenkörper (`Computation`) je um den Betrag 1, weshalb mz nun den Betrag 9 hat. Aus der Abb. 18 ist zu entnehmen, dass die Analyse für jeden untersuchten Fall bis zum Ende des Programms durchgeführt wird. Nach Beendigung der Analyse des Schleifenkörpers wird der Kontrollfluss zum direkten Nachfolger der `PERFORM...UNTIL`-Anweisung zurück gelenkt. Da diese aber die letzte Anweisung im Paragraphen `Control` ist, wird aufgrund der impliziten Reihenfolge von Paragraphen (Fall-Through) im Paragraphen `Computation` weiter gefahren. Die Anweisungen in diesem Paragraphen werden also ein weiteres Mal analysiert. Der bisher geleistete Aufwand steigt deshalb auf den Betrag 11.

Im dritten Fall, wo der Schleifenkörper zweimal analysiert wird, werden die Zwischenresultate aus der erstmaligen Analyse ausgenutzen. Deshalb wird das `PERFORM` und die ihm folgenden Anweisungen nur ein weiteres Mal analysiert und nicht, wie man vermuten könnte, zweimal.

Der Gesamtaufwand der Analyse zur Erzeugung der Program-Slices aller Variablen im Progamm `Factorial-of-6` beträgt somit 18.

5.3 Begründung der Gewichte der Anweisungen in *MiCo*

Bei der Analyse werden jeder Anweisung ein Zustand, der unmittelbar vor ihrer Bearbeitung gilt, übergeben. Die Anweisung modifiziert den erhaltenen Zustand und reicht ihn weiter zur nächsten. Je nach Art der Anweisung sind die Änderungen am Zustand mehr oder weniger aufwendig. Darum werden wir als Grundlage für die Bestimmung des Berechnungsaufwands die Anzahl der im Verlaufe der Analyse erzeugten Zustände heranziehen. Das Rücksetzen von Zuständen wird vernachlässigt, da sie keine neuen, sondern bereits erzeugte Zustände, verwendet.

SKIP $\rightarrow 0$:
Eine `SKIP`-Anweisung verhält sich wie ein Transitknoten. Der ihr übergebene Zustand wird unverändert weitergereicht. Es findet *keine Zustandsänderung* statt.

MOVE $\rightarrow 1$:
Ein neu berechneter Wert für eine Variable wird in den übergebenen Zustand geschrieben. Es muss *ein neuer Zustand* erzeugt werden.

GOTO $\rightarrow 1$:
An jedem Wert der Variablen im übergebenen Zustand werden Modifikationen vorgenommen. Es muss *ein neuer Zustand* erzeugt werden.

IF-THEN-ELSE $\rightarrow 2$:

1. Vor der Analyse der beiden Äste werden die Kontrollabhängigkeiten ergänzt.
2. Die Resultate der unabhängigen Analysen der beiden Äste, welche bis zum Ende des Programms durchgeführt werden, müssen zusammengeführt werden.

Es müssen *zwei neue Zustände* erzeugt werden.

PERFORM $\rightarrow 3$:

1. Vor der Analyse des Schleifenkörpers werden die Kontrollanhängigkeiten um die Beiträge aus der Schleifenbedingung ergänzt.
2. Bei der Analyse des Schleifenkörpers werden die Kontrollanhängigkeiten zusätzlich mit der Marke der `PERFORM`-Anweisung ergänzt.
3. Die Resultate der beiden unabhängigen Analysen (i-mal und $(i-1)$-mal analysieren des Schleifenkörpers) werden zusammengeführt (vergl. Abb. 18).

Es müssen *drei neue Zustände* erzeugt werden[3].

[3]Die `PERFORM`-Anweisungen, welche keine Schleife implementieren, sondern "nur" Subroutinenaufrufe darstellen, sind in der Analyse weniger aufwendig als hier beschrieben. Dies soll uns aber nicht weiter beschäftigen.

5.4 Berechnung des Zeitaufwandes

```
datatype STATEMENT         = MoveStatement
                           | SkipStatement
datatype STATEMENTSEQUENCE = EmptyStatementSequence
                           | StatementSequence of (STATEMENT * STATEMENTSEQUENCE);

datatype BLOCK         = GotoBlock
                       | BasicBlock     of STATEMENTSEQUENCE
                       | ConditionBlock of (BLOCKSEQUENCE * BLOCKSEQUENCE)
                       | LoopBlock      of (BLOCKSEQUENCE * NUMBER)
and      BLOCKSEQUENCE = EmptyBlockSequence
                       | BlockSequence of (BLOCK * BLOCKSEQUENCE);
datatype PROGRAM = Program of BLOCKSEQUENCE;
```

Abbildung 22: Semantische Bereiche zur Berechnung der Masszahl mz

Zur Berechnung von mz muss eine geeignete Darstellung eines *MiCo*-Programms gefunden werden. Als Grundbaustein für die Berechnung des Aufwandes dient ein Block wie oben in Abb. 22 dargestellt. Solche Blöcke lassen sich zu Blocksequenzen zusammenbauen. Es gibt vier verschiedene Arten von Blöcken:

- *Gotoblock:* Er enthält nur eine `GOTO`-Anweisung.
- *Basisblock:* Er enthält ausschliesslich einfache Anweisungen (`SKIP`, `MOVE`).
- *Bedingungsblock:* Er stellt eine bedingte Anweisung dar, welche aus zwei Sequenzen von Blöcken aufgebaut ist, nämlich eine für jeden Ast der bedingten Anweisung.
- *Schleifenblock:* Diese vierte Variante ersetzt ein `PERFORM`, welches eine Sequenz von Blöcken als Schleifenkörper enthält. Diesem Block wird eine Zahl zugeordnet, die angibt, wie oft der Schleifenkörper maximal analysiert werden muss.

Das gesamte Programm kann dann als eine Sequenz von Blöcken (`BLOCKSEQUENCE`) dargestellt werden. Die Reihenfolge der Blöcke innerhalb einer Sequenz hängt vom Kontrollfluss ab, wie er durch die `GOTO`- und `PERFORM`-Anweisungen bestimmt wird.

Als Hilfsfunktion zur Berechnung des Aufwandes für eine Sequenz von einfachen Anweisungen (`SKIP`, `MOVE`) dient die Funktion:

`mz_STATEMENTSEQUENCE: STATEMENTSEQUENCE → ACCU,`

Das Paar `(ACCU * MULT)` hält einen Zustand fest, der unmittelbar vor einem Block gilt. Der erste Eintrag ist ein akkumulierter Wert, der dem geleisteten Aufwand bis zu diesem Programmpunkt entspricht. Der zweite Eintrag ist ein Multiplikator, mit dem der Beitrag des folgenden Blocks zur Masszahl mz multipliziert werden muss. Der Multiplikator hält fest, wie oft die Nachfolger eines Blocks während einer Analyse verwendet werden. Die semantischen Funktion

`mz_PROGRAM: PROGRAM → (ACCU * MULT) → (ACCU * MULT)`

und

`mz_BLOCKSEQUENCE: BLOCKSEQUENCE → (ACCU * MULT) → (ACCU * MULT)`

```
fun mz_BLOCK GotoBlock (accu, mult) = (accu +  mult, mult)
  | mz_BLOCK (BasicBlock          (ss)) (accu, mult)
    = (accu + ((mz_STATEMENTSEQUENCE ss) * mult), mult)
  | mz_BLOCK (ConditionBlock (bs1, bs2)) (accu, mult)
    = let val (x,y) = mz_BLOCKSEQUENCE bs1 (0,mult)
          val (s,t) = mz_BLOCKSEQUENCE bs2 (0,mult)
      in  (accu + (2*mult) + (x+s), y+t) end
  | mz_BLOCK (LoopBlock         (bs, 0)) (accu, mult) = (accu, mult)
  | mz_BLOCK (LoopBlock         (bs, n)) (accu, mult)
    = let val (x,y) = mz_BLOCKSEQUENCE bs (0,mult)
          val (s,t) = mz_BLOCK (LoopBlock (bs, n-1)) (x, y)
      in  (accu + (3*mult) + s, mult+t) end

and mz_BLOCKSEQUENCE (EmptyBlockSequence  ) (accu, mult) = (accu, mult)
  | mz_BLOCKSEQUENCE (BlockSequence (b,bs)) (accu, mult)
    = mz_BLOCKSEQUENCE bs (mz_BLOCK b (accu, mult));

fun mz_PROGRAM (Program bs) (accu, mult) = mz_BLOCKSEQUENCE bs (accu, mult);
```

Abbildung 23: Semantische Funktionen zur Berechnung der Masszahl mz

in Abb. 23 dienen dazu, die Blöcke in der korrekten Reihenfolge zu bearbeiten. Die eigentliche Berechnung der Masszahl wird in der Funktion

`mz_BLOCK: BLOCK → (ACCU * MULT) → (ACCU * MULT)`

durchgeführt.

In den folgenden Unterabschnitten wollen wir die verschiedenen Blöcke genauer untersuchen.

5.4.1 Gotoblock

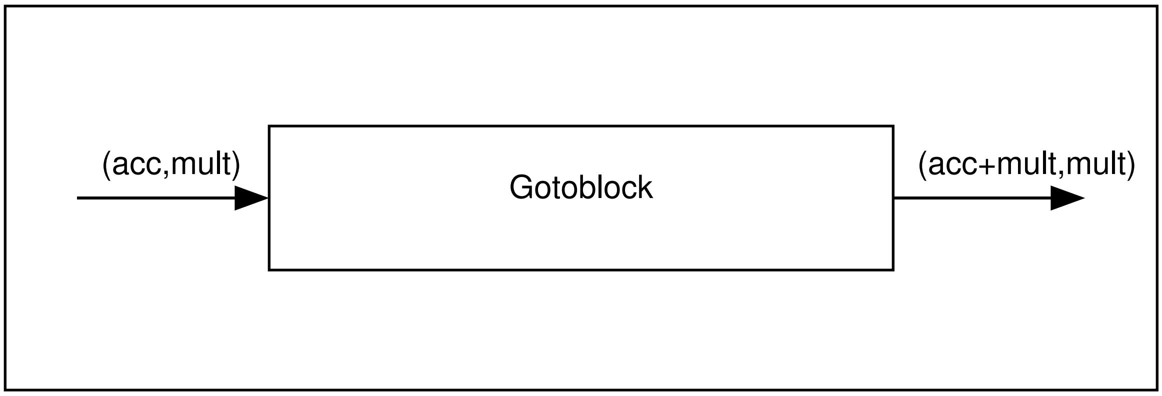

Abbildung 24: Gotoblock

Der Gotoblock erhöht den Wert von mz bei jedem Durchlauf um den Betrag 1. Da dieser Block auch mehrmals analysiert werden kann, was mit dem Multiplikator ausgedrückt wird, muss der Beitrag mit `mult` multipliziert werden.

5.4.2 Basisblock

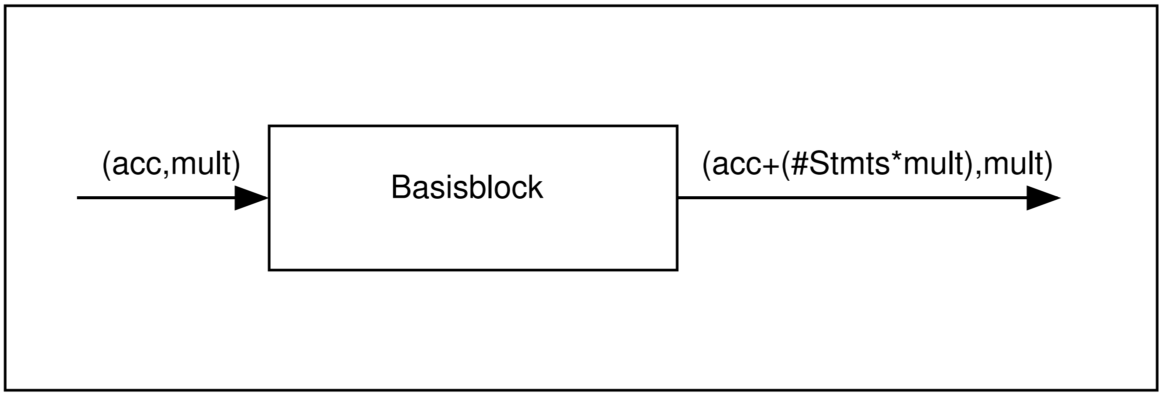

Abbildung 25: Basisblock

Der Basisblock ist der einfachste Blocktyp. Mit Hilfe der Funktion `mz_STATEMENTSEQUENCE` wird die Anzahl der Zuweisungen berechnet. Diese Anzahl wird mit dem alten Multiplikator *mult* multipliziert und zum alten Wert im Akkumulator *accu* addiert. Der Multiplikator wird unverändert weiter gegeben (siehe Abb. 25).

5.4.3 Bedingungsblock

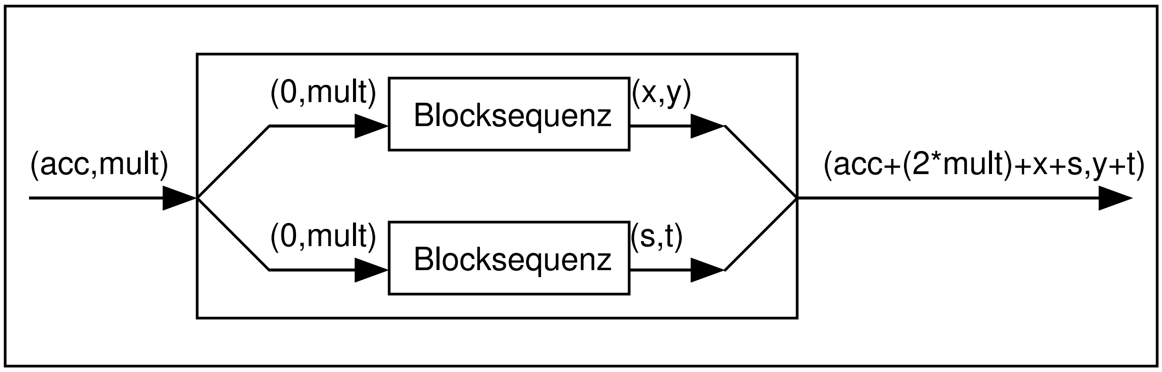

Abbildung 26: Bedingungsblock

Aus der Beschreibung der semantischen Funktion `C` für die bedingte Anweisung (Abb. 14) geht hervor, dass bei der Berechnung des Program-Slices jede der beiden Äste mit der Continuation arbeitet, welche vorgängig der bedingten Anweisung übergeben wurde. Somit wird der Rest des Programms nach der bedingten Anweisung im Verlaufe der Analyse insgesamt zweimal benutzt. Bei der Berechnung der Masszahl *mz* ist dies zu berücksichtigen. Zunächst werden die beiden Blocksequenzen der beiden Äste berechnet; dazu müssen die Akkumulatoren auf Null gesetzt werden, da sonst die "Vorgeschichte" der bedingten Anweisung in beiden Ästen in die Berechnung einfliessen würde. Der Multiplikator wird beibehalten. Nach dem Berechnen der jeweiligen Aufwände für die Blocksequenzen werden die erhaltenen Werte der Akkumulatoren x resp. s zum ursprünglichen Betrag acc addiert. Die Bearbeitung der bedingten Anweisung selbst ist mit 2 gewichtet und wird mit mult multipliziert ebenfalls zu accu addiert. Der neue Wert des Multiplikators ist die Summe der aus den beiden Ästen resultierenden Multiplikatoren. Die Summe deshalb, weil in jeder Blocksequenz möglicherweise erneut Bedingungsblöcke oder Schleifenblöcke enthalten sind, die den Multiplikator verändern können (siehe Abb. 26).

5.4.4 Schleifenblock

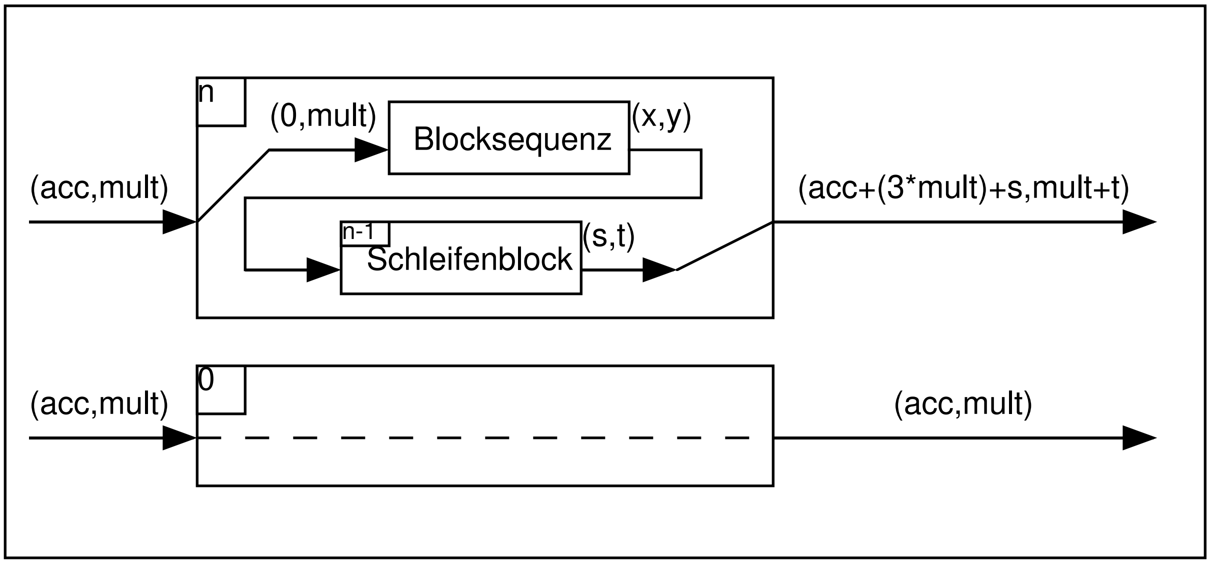

Abbildung 27: Schleifenblock

Betrachten wir die semantische Funktion C für die Varianten der PERFORM-Anweisung (Abb. 17 und 18), so wird der Schleifenkörper während der Analyse maximal so oft durchlaufen, als er Zuweisungen enthält. Für den Fall, dass die PERFORM-Anweisung keine Schleife implementiert, sondern nur ein Subroutienenaufruf darstellt, kann dieser als eine Schleife behandelt werden, die genau einmal analysiert werden muss.

Wie in Abb. 27 ersichtlich ist, wird die Berechnung des Aufwandes einer Schleife mit n Durchläufen auf die Berechnung des Aufwandes einer Schleife mit nur *n-1* Durchläufen zurückgeführt. Für den Fall $n=0$ wird der Schleifenkörper nicht analysiert und trägt darum nichts zum Gesamtaufwand bei.

6 Zusammenfassung und Ausblick

Im vorliegenden Bericht wurde ein Program-Slicer für eine Sprache mit komplexen Kontrollstrukturen basierend auf der denotationellen Semantik beschrieben. Statt, wie in bisherigen Ansätzen die Direct-Style Semantics zu verwenden, haben wir in unserem Ansatz die Continuation-Style Semantics angewandt. Erst dadurch konnte der Kontrollfluss in geeigneter Form repräsentiert und verfolgt werden. Gerade dies hat im vorgestellten Ansatz zentrale Bedeutung, da er darauf beruht, dass die Anweisungen im Programmtext dem Kontrollfluss entsprechend durchlaufen und analysiert werden.

Als Zielsprache wurde *MiCo* verwendet. Sie enthält die essentiellen Kontrollstrukturen von COBOL, mitunter auch das `GOTO` und das `PERFORM`, jedoch vermeidet sie Redundanzen, welche COBOL so umfangreich machen.

Der vorgeschlagene Ansatz berechnet gleichzeitig für sämtliche Variablen eines Programms die zugehörigen Slices. Am Ende des Berichts wurde eine Aufwandabschätzung für das Generieren aller Program-Slices eines *MiCo*-Programms durch unseren Ansatz gegeben. Dabei stellte sich heraus, dass die Aufwände vor allem bei der Analyse von bedingten Anweisungen und Schleifen entstehen.

Das gibt uns Hinweise darauf, wo der Berechnungsaufwand reduziert werden könnte. Zum einen ist zu überlegen, ob bei der Analyse von bedingten Anweisungen nicht darauf verzichtet werden kann, den Rest des Programms nach der bedingten Anweisung zweimal analysieren zu lassen. Zum anderen drängt sich bei der Analyse von Schleifen die Suche nach besseren Methoden auf, um die Anzahl der Analysedurchläufe zu minimieren. Allerdings ist dann abzuklären, welche Einbussen an der Aussagekraft der Program-Slices durch solche "Verbesserungen" in Kauf genommen werden müssen.

Ein Wermuts-Tropfen bleibt allerdings noch. Unser Ansatz lässt sich in der vorliegenden Form leider noch nicht direkt auf COBOL übertragen: erstens aufgrund der oben genannten Aufwände bei der Analyse und zweitens, was entscheidenter ist, weil in absehbarer Zeit keine vollständige Beschreibung der Sprache COBOL in denotationeller Semantik verfügbar sein wird. Allerdings können die Erfahrungen, die aus der Arbeit mit *MiCo* gewonnen wurden, für die Entwicklung eines COBOL-spezifischen Program-Slicers genutzt werden. Ein solcher Program-Slicer wird aber auf eine wohldefinierte formale Beschreibung der Sprachkonstrukte einstweilen verzichten müssen.

A Program-Slicing des *MiCo*-Programms in Abb. 19 in Abschnitt 4.5

A.1 Das zu analysierende *MiCo*-Programm `Example`

```
PROGRAM Example .
  DATA DIVISION .
    i, k, v, x, y, z : NUMERIC
  END
  PROCEDURE DIVISION .
    PARAGRAPH Initialization .         (*>1<*)
      MOVE 0 TO i .                    (*>2<*)
      MOVE 0 TO v .                    (*>3<*)
      MOVE 1 TO x .                    (*>4<*)
      MOVE 2 TO y .                    (*>5<*)
      MOVE 3 TO z .                    (*>6<*)
      PERFORM WhileLoop UNTIL i > 10 . (*>7<*)
      MOVE v TO k .                    (*>8<*)
      GOTO Exit                        (*>9<*)
    END .
    PARAGRAPH WhileLoop .             (*>10<*)
      MOVE z TO v .                   (*>11<*)
      MOVE y TO z .                   (*>12<*)
      MOVE x TO y .                   (*>13<*)
      MOVE i + 1 TO i                 (*>14<*)
    END .
    PARAGRAPH Exit .                  (*>15<*)
      SKIP                            (*>16<*)
    END
  END
END
```

Abbildung 28: Das *MiCo*-Programm `Example`

A.2 Die Zustandsübergänge im Verlauf der Analyse

- **Initialisieren**

  $s_0 = ((\{i \to \{\}, v \to \{\}, k \to \{\}, x \to \{\}, y \to \{\}, z \to \{\}), \{\})$

- **MOVE 0 TO i**

  $s_1 = ((\{i \to \{2\}, v \to \{\}, k \to \{\}, x \to \{\}, y \to \{\}, z \to \{\}), \{\})$

- **MOVE 0 TO v**

  $s_2 = ((\{i \to \{2\}, v \to \{3\}, k \to \{\}, x \to \{\}, y \to \{\}, z \to \{\}), \{\})$

- **MOVE 1 TO x**

  $s_3 = ((\{i \to \{2\}, v \to \{3\}, k \to \{\}, x \to \{4\}, y \to \{\}, z \to \{\}), \{\})$

- **MOVE 2 TO y**

  $s_4 = ((\{i \to \{2\}, v \to \{3\}, k \to \{\}, x \to \{4\}, y \to \{5\}, z \to \{\}), \{\})$

- **MOVE 3 TO z**

  $s_5 = ((\{i \to \{2\}, v \to \{3\}, k \to \{\}, x \to \{4\}, y \to \{5\}, z \to \{6\}), \{\})$

- **PERFORM WhileLoop UNTIL i > 10**

  Bemerkung: Die Schleife muss unabhängig voneinander für null-, ein-, zwei-, drei- und viermaliges Durchlaufen analysiert werden. Anschliessend werden sie zusammengeführt.

Nullmal

1. **MOVE v TO k**

   $s_6 = ((\{i \to \{2\}, v \to \{3\}, k \to \{3, 8\}, x \to \{4\}, y \to \{5\}, z \to \{6\}), \{\})$
2. **GOTO Exit**

   $s_7 = ((\{i \to \{2, 9, 15\}, v \to \{3, 9, 15\}, k \to \{3, 8, 9, 15\},$
   $x \to \{4, 9, 15\}, y \to \{5, 9, 15\}, z \to \{6, 9, 15\}), \{\})$
3. **SKIP**

   $s_8 = ((\{i \to \{2, 9, 15\}, v \to \{3, 9, 15\}, k \to \{3, 8, 9, 15\},$
   $x \to \{4, 9, 15\}, y \to \{5, 9, 15\}, z \to \{6, 9, 15\}), \{\})$

Erstesmal

1. **Kontrollabhängigkeiten bestimmen**

   $s_9 = ((\{i \to \{2\}, v \to \{3\}, k \to \{\},$
   $x \to \{4\}, y \to \{5\}, z \to \{6\}), \{2, 7\})$
2. **MOVE z TO v**

   $s_{10} = ((\{i \to \{2\}, v \to \{2, 6, 7, 11\}, k \to \{\},$
   $x \to \{4\}, y \to \{5\}, z \to \{6\}), \{2, 7\})$
3. **MOVE y TO z**

   $s_{11} = ((\{i \to \{2\}, v \to \{2, 6, 7, 11\}, k \to \{\},$
   $x \to \{4\}, y \to \{5\}, z \to \{2, 5, 7, 12\}), \{2, 7\})$
4. **MOVE x TO y**

   $s_{12} = ((\{i \to \{2\}, v \to \{2, 6, 7, 11\}, k \to \{\},$
   $x \to \{4\}, y \to \{2, 4, 7, 13\}, z \to \{2, 5, 7, 12\}), \{2, 7\})$

5. **MOVE i+1 TO i**

$s_{13} = ((\{i \to \{2,7,14\}, v \to \{2,6,7,11\}, k \to \{\},$
$x \to \{4\}, y \to \{2,4,7,13\}, z \to \{2,5,7,12\}), \{2,7\})$

6. **Kontrollabhängigkeiten zurücksetzen**

$s_{14} = ((\{i \to \{2,7,14\}, v \to \{2,6,7,11\}, k \to \{\},$
$x \to \{4\}, y \to \{2,4,7,13\}, z \to \{2,5,7,12\}), \{\})$

7. **MOVE v TO k**

$s_{15} = ((\{i \to \{2,7,14\}, v \to \{2,6,7,11\}, k \to \{2,6,7,8,11\},$
$x \to \{4\}, y \to \{2,4,7,13\}, z \to \{2,5,7,12\}), \{\})$

8. **GOTO Exit**

$s_{16} = ((\{i \to \{2,7,9,14,15\}, v \to \{2,6,7,9,11,15\}, k \to \{2,6,7,8,9,11,15\},$
$x \to \{4,9,15\}, y \to \{2,4,7,9,13,15\}, z \to \{2,5,7,9,12,15\}), \{\})$

9. **SKIP**

$s_{17} = ((\{i \to \{2,7,9,14,15\}, v \to \{2,6,7,9,11,15\}, k \to \{2,6,7,8,9,11,15\},$
$x \to \{4,9,15\}, y \to \{2,4,7,9,13,15\}, z \to \{2,5,7,9,12,15\}), \{\})$

Zweitesmal

1. **Kontrollabhängigkeiten bestimmen**

$s_{18} = ((\{i \to \{2,7,14\}, v \to \{2,6,7,11\},$
$k \to \{\}, x \to \{4\},$
$y \to \{2,4,7,13\}, z \to \{2,5,7,12\}), \{2,7,14\})$

2. **MOVE z TO v**

$s_{19} = ((\{i \to \{2,7,14\}, v \to \{2,5,7,11,12,14\},$
$k \to \{\}, x \to \{4\},$
$y \to \{2,4,7,13\}, z \to \{2,5,7,12\}), \{2,7,14\})$

3. **MOVE y TO z**

$s_{20} = ((\{i \to \{2,7,14\}, v \to \{2,5,7,11,12,14\},$
$k \to \{\}, x \to \{4\},$
$y \to \{2,4,7,13\}, z \to \{2,4,7,12,13,14\}), \{2,7,14\})$

4. **MOVE x TO y**

$s_{21} = ((\{i \to \{2,7,14\}, v \to \{2,5,7,11,12,14\},$
$k \to \{\}, x \to \{4\},$
$y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{2,7,14\})$

5. **MOVE i+1 TO i**

$s_{22} = ((\{i \to \{2,7,14\}, v \to \{2,5,7,11,12,14\},$
$k \to \{\}, x \to \{4\},$
$y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{2,7,14\})$

6. **Kontrollabhängigkeiten zurücksetzen**

$s_{23} = ((\{i \to \{2,7,14\}, v \to \{2,5,7,11,12,14\},$
$k \to \{\}, x \to \{4\},$
$y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{\})$

7. **MOVE v TO k**

$s_{24} = ((\{i \to \{2,7,14\}, v \to \{2,5,7,11,12,14\},$
$k \to \{2,5,7,8,11,12,14\}, x \to \{4\},$
$y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{\})$

8. **GOTO Exit**

$$s_{25} = ((\{i \to \{2,7,9,14,15\}, v \to \{2,5,7,9,11,12,14,15\},$$
$$k \to \{2,5,7,8,9,11,12,14,15\}, x \to \{4,9,15\},$$
$$y \to \{2,4,7,9,13,14,15\}, z \to \{2,4,7,9,12,13,14,15\}), \{\})$$

9. **SKIP**

$$s_{26} = ((\{i \to \{2,7,9,14,15\}, v \to \{2,5,7,9,11,12,14,15\},$$
$$k \to \{2,5,7,8,9,11,12,14,15\}, x \to \{4,9,15\},$$
$$y \to \{2,4,7,9,13,14,15\}, z \to \{2,4,7,9,12,13,14,15\}), \{\})$$

Drittesmal

1. **Kontrollabhängigkeiten bestimmen**

$$s_{27} = ((\{i \to \{2,7,14\}, v \to \{2,5,7,11,12,14\},$$
$$k \to \{\}, x \to \{4\},$$
$$y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{2,7,14\})$$

2. **MOVE z TO v**

$$s_{28} = ((\{i \to \{2,7,14\}, v \to \{2,4,7,11,12,13,14\},$$
$$k \to \{\}, x \to \{4\},$$
$$y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{2,7,14\})$$

3. **MOVE y TO z**

$$s_{29} = ((\{i \to \{2,7,14\}, v \to \{2,4,7,11,12,13,14\},$$
$$k \to \{\}, x \to \{4\},$$
$$y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{2,7,14\})$$

4. **MOVE x TO y**

$$s_{30} = ((\{i \to \{2,7,14\}, v \to \{2,4,7,11,12,13,14\},$$
$$k \to \{\}, x \to \{4\},$$
$$y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{2,7,14\})$$

5. **MOVE i+1 TO i**

$$s_{31} = ((\{i \to \{2,7,14\}, v \to \{2,4,7,11,12,13,14\},$$
$$k \to \{\}, x \to \{4\},$$
$$y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{2,7,14\})$$

6. **Kontrollabhängigkeiten zurücksetzen**

$$s_{32} = ((\{i \to \{2,7,14\}, v \to \{2,4,7,11,12,13,14\},$$
$$k \to \{\}, x \to \{4\},$$
$$y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{\})$$

7. **MOVE v TO k**

$$s_{33} = ((\{i \to \{2,7,14\}, v \to \{2,4,7,11,12,13,14\},$$
$$k \to \{2,4,7,8,11,12,13,14\}, x \to \{4\},$$
$$y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{\})$$

8. **GOTO Exit**

$$s_{34} = ((\{i \to \{2,7,9,14,15\}, v \to \{2,4,7,9,11,12,13,14,15\},$$
$$k \to \{2,4,7,8,9,11,12,13,14,15\}, x \to \{4,9,15\},$$
$$y \to \{2,4,7,9,13,14,15\}, z \to \{2,4,7,9,12,13,14,15\}), \{\})$$

9. **SKIP**

$$s_{35} = ((\{i \to \{2,7,9,14,15\}, v \to \{2,4,7,9,11,12,13,14,15\},$$
$$k \to \{2,4,7,8,9,11,12,13,14,15\}, x \to \{4,9,15\},$$
$$y \to \{2,4,7,9,13,14,15\}, z \to \{2,4,7,9,12,13,14,15\}), \{\})$$

Viertesmal

1. **Kontrollabhängigkeiten bestimmen**
   $s_{36} = ((\{i \to \{2,7,14\}, v \to \{2,4,7,11,12,13,14\},$
   $k \to \{\}, x \to \{4\},$
   $y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{2,7,14\})$
2. **MOVE z TO v**
   $s_{37} = ((\{i \to \{2,7,14\}, v \to \{2,4,7,11,12,13,14\},$
   $k \to \{\}, x \to \{4\},$
   $y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{2,7\})$
3. **MOVE y TO z** $s_{38} = ((\{i \to \{2,7,14\}, v \to \{2,4,7,11,12,13,14\},$
   $k \to \{\}, x \to \{4\},$
   $y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{2,7\})$
4. **MOVE x TO y** $s_{39} = ((\{i \to \{2,7,14\}, v \to \{2,4,7,11,12,13,14\},$
   $k \to \{\}, x \to \{4\},$
   $y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{2,7\})$
5. **MOVE i+1 TO i** $s_{40} = ((\{i \to \{2,7,14\}, v \to \{2,4,7,11,12,13,14\},$
   $k \to \{\}, x \to \{4\},$
   $y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{2,7\})$
6. **Kontrollabhängigkeiten zurücksetzen**
   $s_{41} = ((\{i \to \{2,7,14\}, v \to \{2,4,7,11,12,13,14\},$
   $k \to \{\}, x \to \{4\},$
   $y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{\})$
7. **MOVE v TO k**
   $s_{42} = ((\{i \to \{2,7,14\}, v \to \{2,4,7,11,12,13,14\},$
   $k \to \{2,4,7,8,11,12,13,14\}, x \to \{4\},$
   $y \to \{2,4,7,13,14\}, z \to \{2,4,7,12,13,14\}), \{\})$
8. **GOTO Exit**
   $s_{43} = ((\{i \to \{2,7,9,14,15\}, v \to \{2,4,7,9,11,12,13,14,15\},$
   $k \to \{2,4,7,8,9,11,12,13,14,15\}, x \to \{4,9,15\},$
   $y \to \{2,4,7,9,13,14,15\}, z \to \{2,4,7,9,12,13,14,15\}), \{\})$
9. **SKIP**
   $s_{44} = ((\{i \to \{2,7,9,14,15\}, v \to \{2,4,7,9,11,12,13,14,15\},$
   $k \to \{2,4,7,8,9,11,12,13,14,15\}, x \to \{4,9,15\},$
   $y \to \{2,4,7,9,13,14,15\}, z \to \{2,4,7,9,12,13,14,15\}), \{\})$

Zusammenführen

1. Resultat aus nullmaligem Durchlaufen der Schleife zusammenfügen mit dem Resultat aus einmaligem Durchlaufen der Schleife:
   $$\begin{aligned} s_{45} &= s_8 \sqcup s_{17} \\ &= ((\{i \to \{2,7,9,14,15\}, v \to \{2,3,6,7,9,11,15\}, \\ &\quad k \to \{2,3,6,7,8,9,11,15\}, x \to \{4,9,15\}, \\ &\quad y \to \{2,4,5,7,9,13,15\}, z \to \{2,5,6,7,9,12,15\}), \{\}) \end{aligned}$$
2. Zwischenresultat zusammenfügen mit dem Resultat aus zweimaligem Durchlaufen der Schleife:
   $$\begin{aligned} s_{46} &= s_{45} \sqcup s_{26} \\ &= ((\{i \to \{2,7,9,14,15\}, v \to \{2,3,5,6,7,9,11,12,14,15\}, \end{aligned}$$

$k \to \{2, 3, 5, 6, 7, 8, 9, 11, 12, 14, 15\}, x \to \{4, 9, 15\},$
$y \to \{2, 4, 5, 7, 9, 13, 14, 15\}, z \to \{2, 4, 5, 6, 7, 9, 12, 13, 14, 15\}), \{\})$

3. Zwischenresultat zusammenfügen mit dem Resultat aus dreimaligem Durchlaufen der Schleife:

$$\begin{aligned} s_{47} &= s_{46} \sqcup s_{35} \\ &= ((\{i \to \{2, 7, 9, 14, 15\}, v \to \{2, 3, 4, 5, 6, 7, 9, 11, 12, 13, 14, 15\}, \\ &\qquad k \to \{2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 14, 15\}, x \to \{4, 9, 15\}, \\ &\qquad y \to \{2, 4, 5, 7, 9, 13, 14, 15\}, z \to \{2, 4, 5, 6, 7, 9, 12, 13, 14, 15\}), \{\}) \end{aligned}$$

4. Zwischenresultat zusammenfügen mit dem Resultat aus viermaligem Durchlaufen der Schleife zum Endresultat:

$$\begin{aligned} s_{48} &= s_{47} \sqcup s_{44} \\ &= ((\{i \to \{2, 7, 9, 14, 15\}, v \to \{2, 3, 4, 5, 6, 7, 9, 11, 12, 13, 14, 15\}, \\ &\qquad k \to \{2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15\}, x \to \{4, 9, 15\}, \\ &\qquad y \to \{2, 4, 5, 7, 9, 13, 14, 15\}, z \to \{2, 4, 5, 6, 7, 9, 12, 13, 14, 15\}), \{\}) \end{aligned}$$

Literatur

[AH90] H. Agrawal and J.R. Horgan. Dynamic Program Slicing. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 246–256. ACM Press, June 1990.

[AS88] K. Alber and W. Struckmann. *Einführung in die Semantik von Programmiersprachen.* BI Wissenschaftsverlag Mannheim, 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, 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, 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, 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, 1993.

[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.

[HR92] S. Horwitz and T. Reps. The Use of Program Dependence Graphs in Software Engineering. *IEEE 14th International Conference On Software Engineering 1992*, pages 392–411, May 1992.

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

[LR87] Hareton K.N. Leung and Hassan K. Reghbati. Comments on Program Slicing. *IEEE Transactions on Software Engineering*, 13(12):1370–1371, December 1987.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[NN92] H.R. Nielson and F. Nielson. *Semantics with Applications.* John Wiley & Sons, 1992.

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

[Sch86] D.A. Schmidt. *Denotational Semantics - A Methodology for Language Development.* WM. C. Brown Publishers, 1986.

[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.

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

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