Ludwig-Maximilians-Universität München

Institut für Informatik

Seminararbeit

im Studiengang Medieninformatik

Thema: Datenflussanalyse - Klassisch

eingereicht von: Lorenz Schauer

eingereicht am: 4. Juli 2009

Betreuer: Prof. Martin Hofmann

Inhaltsverzeichnis

1 Einleitung

In dem zusammengesetzten Wort „Datenflussanalyse“ stecken bereits zwei wesentliche Charakteristika dieses weitumfassenden Themengebietes.
Zum einen geht es um diejenigen Daten, welche Informationen halten, die für den Programmfluss entscheidend sind beziehungsweise welche eine Aussage über die Ausführung des vorliegenden Programms treffen können.
Zum anderen ist es eine Form der Anaylse, welche vorwiegend in der theoretischen Informatik Verwendung findet. Die Datenflussanalyse hat es sich hierbei zur Aufgabe gemacht, die Zusammenhänge zwischen den, an einzelnen Stellen im Programm berechneten Werten, zu ermitteln.
Da ein Programm in seiner Ausführung verschiedene Wege gehen kann, beispielsweise durch Fallunterscheidungen, werden bei der Analyse des jeweiligen Datenflusses alle Programmabläufe berücksichtigt, die eventuell abgearbeitet werden können.
Die Datenflussanalyse bildet zusammen mit der Kontrollflussanalyse ein mächtiges Werkzeug, um beim Kompilieren des Programms Optimierungen am Quellcode vorzunehmen. Ein weiteres Ziel besteht darin, Datenflussprobleme zu formulieren um sie später mit Standardverfahren zu lösen. Dabei ist es unbedingt erforderlich den vorgegebenen Code beziehungsweise das zugrunde liegende Programmstück zu verstehen und zu hinterfragen. Typische Fragestellungen können sein:

- Was geanu wird in dem Programmabschnitt berechnet?
- Wie und wo wird eine bestimmte Variable berechnet?
- Von welchen Variablen hängt ein Wert an einer betrachteten Programmstelle ab?
- Welche Werte kann die Variable annehmen?
- Welche Befehle beinflussen welche Variable zu welchem Zeitpunkt?

Eine generelle Vorgehensweise in der Datenflussanalyse gliedert sich in folgende schritte:
Zuerst muss der Programmcode betrachtet und verstanden werden. Hierbei eignet sich ein so genannter Flussgraph, der die gegebenen Aussagen und deren Berechnungen grafisch darstellt. Darauf aufbauend sollten die Datenflussprobleme erkannt und die korrespondierenden Gleichungen aufgestellt werden. Mit Hilfe dieser Gleichungen kann dann der zugrunde liegende Code optimiert und gegebenenfalls die damit verbundenen Probleme gelöst werden. Praktische Anwendung in der heutigen Zeit findet die Datenflussanalyse im Compilerbau, wo sie dazu benutzt wird, den zu übersetzenden Code zu optimieren bevor er in Maschinensprache umgesetzt wird. Das bringt sowohl Speicherplatzvorteile als auch höhere Performanz und somit geringere Kosten. Dies ist nur ein Beispiel dafür, dass das behandelte Thema nicht nur

als wissenschaftliches und theoretisches Forschungsgebiet verstanden werden soll, sondern auch eine Grundlage zur Generierung von konkretem wirtschaftlichen Nutzen bildet.

Da es eine Reihe unterschiedlicher Arten der Datenflussanalyse gibt werden in dieser Seminararbeit die einzelnen Konzepte vorgestellt und jeweils an einem konkreten Beispiel veranschaulicht. Im speziellen werden nun die Methoden der **Intraprozeduralen Analyse** erläutert, wie sie auch im Buch *Principles of Program Analysis* von Flemming Nielson, Hanne Riis Nielson, Chris Hankin (2005) beschrieben sind. Im letzten Abschnitt wird noch auf die theoretischen Konzepte eingegangen, welche als Grundlage der *Live Variables Analysis* dienen.

2 Die intraprozedurale Analyse

Die Autoren Flemming, Hanne und Chris beschreiben in ihrem erwähnten Buch verschiedene Analysearten, die sie alle unter dem Begriff der „intraprozeduralen Analyse“ zusammenfassen.
Der nicht ganz gängige Begriff „intraprozedural“ stammt vom lateinischen „intra“ *(innerhalb)* und dem aus der Informatik bekanntem Begriff der „Prozedur“. Dadurch lässt sich erkennen, dass diese Form der Analyse sich mit dem Datenfluss innerhalb einer Prozedur beschäftigt. Dabei betrachtet man nur Definitionen und Variablenbelegungen, die auch innerhalb der untersuchten Prozedur konkretisiert werden und kümmert sich nicht darum, was mit den Variablen vor und nach der Prozedur passiert. Hierbei werden meistens Schleifen betrachtet, da sich dann der zu analysierende Code solange wiederholt bis eine entsprechende Bedingung nicht mehr erfüllt ist und somit die beteiligten Variablen innerhalb der Schleife ständig benötigt und dabei gegebenenfalls verändert werden.

In der intraprozeduralen Analyse wird eine Betrachtung bezüglich der Flussrichtung wichtig. Man unterscheidet zwischen Vorwärts- und Rückwärtsanalyse.
Die Vorwärtsanalyse beobachtet eine Aussage in der Gegenwart und analysiert, was mit ihr in der Zukunft, beziehungsweise im weiteren Programmverlauf passiert.
Die Rückwärtsanalyse überprüft dagegen, wo diese Aussage im vorangegangenen Code beschrieben wurde.
Die mathematische Beschreibung für die Betrachtung der Flussrichtung wird später in diesem Kapitel mit der Definition der Flussfunktion konkretisiert.

Um verschiedene Arten der intraprozeduralen Analyse auf den nachfolgenden Seiten erläutern zu können, bedarf es nun einiger Definitionen und Beschreibungen von Formeln, die für eine Untersuchung eines bestimmten Pro-

grammstücks in der Sprache `WHILE` von essentieller Bedeutung sind.

2.1 Formeln und Definitionen

Zuerst wird die Initialfunktion definiert, welche das Anfangslabel des betrachteten Programms zurückgibt:

$$\text{init: } \mathbf{Stmt} \rightarrow \mathbf{Lab}$$

Angewandt auf die standard `WHILE`-Konstrukte liefert diese Funktion folgende Werte:

$$\begin{aligned}
&init([x := a]^l) = l \\
&init([skip]^l) = l \\
&init(S_1; S_2) = init(S_1) \\
&init(if [b]^l then S_1 else S_2) = l \\
&init(while [b]^l do S) = l
\end{aligned}$$

Es ist zu erwähnen, dass hier S für eine Aussage (*engl.: Statement*) und b für eine boolesche Bedingung steht. Zwei Aussagen, welche mit einem Strichpunkt getrennt sind, lassen sich als eine Sequenz im Programm auffassen. Man erkennt, dass die Funktion immer das Label l des ersten Ausdrucks im Programm liefert.
Neben der Initialsfunktion wird auch eine Endfunktion benötigt, welche das Label des letzten Blocks im Programm zurückgibt. Diese ist quasi die umgekehrte Initialsfunktion und lässt sich wie folgt beschreiben:

$$\text{final: } \mathbf{Stmt} \rightarrow \mathrm{P}(\mathbf{Lab})$$

Und die Anwendung auf die entsprechenden Aussagen liefert:

$$\begin{aligned}
&final([x := a]^l) = \{l\} \\
&final([skip]^l) = \{l\} \\
&final(S_1; S_2) = final(S_2) \\
&final(if [b]^l then S_1 else S_2) = final(S_1) \cup final(S_2) \\
&final(while [b]^l do S) = \{l\}
\end{aligned}$$

Um in einem Programm Elementaraussagen betrachten zu können, wird der Code in einzelne Blöcke mit Hilfe der Blockfunktion zerlegt:

$$\text{blocks: } \mathbf{Stmt} \rightarrow \mathrm{P}(\mathbf{Blocks})$$

Da die so genannten Blöcke nur Zuweisungen sowie boolesche Bedingungen und die *skip*-Anweisung enthalten sollen, liefert die blocks-Funktion in Bezug auf die Standardaussagen folgende Gleichungen:

$$
\begin{aligned}
&blocks([x := a]^l) = \{[x := a]^l\} \\
&blocks([skip]^l) = \{[skip]^l\} \\
&blocks(S_1; S_2) = blocks(S_1) \cup blocks(S_2) \\
&blocks(if[b]^l then S_1 else S_2) = \{[b]^l\} \cup blocks(S_1) \cup blocks(S_2) \\
&blocks(while[b]^l do S) = \{[b]^l\} \cup blocks(S)
\end{aligned}
$$

Als letzte Hilfsfunktion wird noch die *labels*-Funktion bestimmt, welche das Label einer Aussage zurückgibt. Diese Funktion schließt auch *init(S)* und *final(S)* mit ein:

$$\text{labels: } \mathbf{Stmt} \rightarrow \mathrm{P}(\mathbf{Lab})$$

Dabei ist $label(S) = \{l | [B]^l \in blocks(S)\}$

In Bezug auf einen Ausdruck sagt man, er sei *label konsistent*, wenn gilt:

$$[B_1]^l, [B_2]^l \in blocks(S) <=> B_1 = B_2$$

Mit Hilfe dieser mathematischen Beschreibungen kann jetzt die oben erwähnte Flussfunktion definiert werden. Zuerst für die Vorwärtsanalyse:

$$flow : \mathbf{Stmt} \rightarrow P(\mathbf{Lab} \times \mathbf{Lab})$$

$$
\begin{aligned}
&flow([x := a]^l) = \{\} \\
&flow([skip]^l) = \{\} \\
&flow(S_1; S_2) = flow(S_1) \cup flow(S_2) \cup \{(l, init(S_2)) | l \in final(S_1)\} \\
&flow(if[b]^l then S_1 else S_2) = flow(S_1) \cup flow(S_2) \cup \{(l, init(S_1)), (l, init(S_2))\} \\
&flow(while[b]^l do S) = flow(S) \cup \{(l, init(S))\} \cup \{(l', l) | l' \in final(S)\}
\end{aligned}
$$

Es sollte klar sein, dass die Flussfunktion auf Zuweisungen und „skip“ keine Werte liefern kann, da innerhalb eines Blocks kein Datenfluss stattfindet. Der Fluss besteht zwischen den Labels, was bei der Sequenz ersichtlich wird. Hier liefet die Funktion den Fluss vom Ende des Labels in dem S_1 steht hin zum Anfang von dem Label, in welchem S_2 beschrieben wird.
Die Funktion der Rückwärtsanalyse $flow^R$ steht im direkten Zusammenhang mit $flow$ und ist definiert als:

$$flow^R(S) = \{(l, l') | (l', l) \in flow(S)\}$$

Ein Flussgraph kann ein gutes Hilfsmittel sein, um den Datenfluss zu verstehen und damit beschreiben zu können. Es soll nun mit Hilfe der oben definierten Flussfunktionen ein solcher Flussgraph entstehen. Dazu betrachte man folgendes Beispiel:

$$[z := 1]^1; while[x > 0]^2 do([z := z * y]^3; [x := x - 1]^4)$$

Die Flussfunktion der Vorwärtsanalyse liefert nun bezüglich des Beispiels folgende Menge:

$$\{(1, 2), (2, 3), (3, 4), (4, 2)\}$$

Diese Menge sagt aus, dass es jeweils ein Fluss zwischen den Labels von 1 bis 4 gibt, aber wegen der Schleife auch von 4 zu 2. Damit lässt sich der Flussgraph zeichnen und ist in der nachfolgenden Grafik dargestellt.

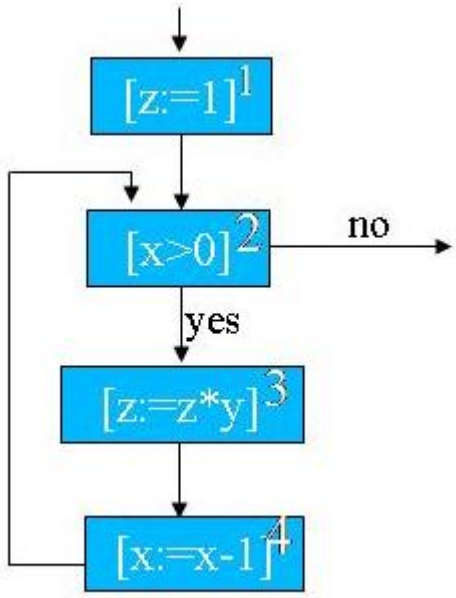

Abbildung 1: Flussgraph

Es ist zu erkennen, dass bei Erfüllung der booleschen Bedingung der Programmpfad nach unten hin fortgesetzt wird und schließlich wieder vor der Schleife landet. Ist die Bedingung nicht mehr erfüllt, so läuft das Programm auf den mit „no“ beschrifteten Pfeil weiter.

Wenn ein komplettes Programm betrachtet wird, so soll die Notation für Aussage, Label und Block erhalten bleiben, aber der korrespondierende Buchstabe erhält als Index einen Stern. Dieser besagt, dass immer die höchste

Instanz, also das Wurzelelement untersucht wird. Beispielsweise wird die oberste Aussage als S_* ausgedrückt.

An dieser Stelle sollen auch gleich noch erwähnt werden:

- Var_* steht für die Variablen, welche in S_* vorkommen
- $AExp_*$ steht für einen nicht trivialen arithmetischen Ausdruck
- $AExp(a)$ repräsentiert einen gegebenen nicht trivialen arithmetischen Ausdruck von a

Auf Grundlage dieser Definitionen kann nun eine strukturierte Betrachtung des vorliegenden Codes durchgeführt werden. Mit Hilfe der Flussfunktionen lässt sich der Datenfluss mathematisch und durch den daraus resultierenden Flussgraphen auch graphisch zeigen. Die *labels* und *blocks* Funktionen zerlegen das Programm in Blöcke mit einem eindeutigem Label. In den anstehenden Unterkapiteln werden jetzt die Konzepte der verschiedenen Analysearten beschrieben und erklärt.

2.2 Available Expressions Analysis

Die „Analyse der verfügbaren Ausdrücke“ beschäftigt sich, wie der Name schon sagt, mit der Verfügbarkeit einzelner Ausdrücke an der jeweils betrachteten Programmstelle. Ein Ausdruck ist an einem Programmpunkt immer dann verfügbar, wenn er vorher generiert und bis zu dem betrachteten Punkt nicht gelöscht beziehungsweise keine seiner Variablen überschrieben wurde. Somit stellt diese Form eine Vorwärtsanalyse dar mit der Hauptaufgabe den vorliegenden Quellcode zu optimieren. Die Informationen, die hieraus abgeleitet werden können dazu benutzt werden eine wiederholte Definition eines bereits verfügbaren Ausdruckes zu verhindern.
Um nun der Frage nachgehen zu können, welche Ausdrücke an welcher Stelle verfügbar sind, müssen zunächst die *gen* und *kill* Funktionen der *Available Expressions Analysis* aufgestellt werden.
Die *gen*-Funktion drückt aus, dass ein Ausdruck im Block generiert und gleichzeitig keine Variable des Ausdrucks geändert wurde:

$$gen_{AE} : Blocks_* \rightarrow P(AExp_*)$$

Die *kill*-Funktion hingegen beschreibt, dass eine Variable des Ausdrucks innerhalb des betrachteten Blocks verändert wurde:

$$kill_{AE} : Blocks_* \rightarrow P(AExp_*)$$

Angewant auf Zuweisung, skip und boolesche Bedingung ergeben diese Funktionen folgende Gleichungen:

$$kill_{AE}([x := a]^l) = \{a' \in AExp_* | x \in FV(a')\}$$
$$kill_{AE}([skip]^l) = \{\}$$
$$kill_{AE}([b]^l) = \{\}$$

$$gen_{AE}([x := a]^l) = \{a' \in AExp(a) | x \notin FV(a')\}$$
$$gen_{AE}([skip]^l) = \{\}$$
$$gen_{AE}([b]^l) = AExp(b)$$

Falls die *kill*-Funktion einen Wert zurückliefert, heißt das für den zurückgegebenen Ausdruck, dass dieser für den weiteren Programmverlauf nicht mehr verfügbar ist. In dieser Seminararbeit wird dies stets mit dem eingedeutschten Begriff „killen“ ausgedrückt.
Es sollte deutlich werden, dass eine *kill*-Funktion auf eine boolesche Bedingung oder auf einen skip-Ausdruck eine leere Menge zurückliefert, also keinen Ausdruck „killen“ kann, wo hingegen bei Anwendung auf eine Zuweisung, bei welcher beispielsweise der Wert von a auf eine Variable x geschrieben wird, jeder vorhergehende Ausdruck, bei dem die Variable x vorkommt, als nicht verfügbar eingestuft wird.
Ähnlich verhält es sich mit der *gen*-Funktion. Diese hat auch auf den skip-Ausdruck angewant keinen Einfluss, liefert also eine leere Menge zurück. Sie generiert jedoch bei einer Zuweisung und bei einer booleschen Bedingung einen neuen verfügbaren Ausdruck, der dann solange verfügbar bleibt, bis er im weiteren Verlauf durch die *kill*-Funktion wieder „gekilled“ wird.
Mit diesen Beschreibungen lassen sich jetzt auch die Eingangs- und Ausgangsfunktionen für die eigentliche Analyse definieren:

$$AE_{entry}(l) = \left\{ \begin{array}{c} \{\} \quad falls \quad l = init(S_*) \\ \bigcap(AE_{exit}(l')|(l', l) \in flow(S_*)) \end{array} \right\}$$
$$AE_{exit}(l) = (AE_{entry}(l) \backslash kill_{AE}(B^l)) \cup gen_{AE}(B^l) \text{wo B}^l \in blocks(S_*)$$

Die *entry* und *exit* Funktionen werden stets rekursiv aufgerufen und durchlaufen so das zu überprüfende Programm. Aus den Definitionen wird ersichtlich, dass ein Ausdruck, der am Anfang eines Blocks verfügbar war dies auch am Ausgang wieder ist, wenn er nicht innerhalb des Blocks „gekilled“, sprich neu zugewiesen wurde. Die Eingangsfunktion ist am Programmanfang natürlich leer und liefert ansonsten denjenigen Ausdruck zurück, der am Ausgang des vorhergehenden Blocks verfügbar war.

Um diese Funktionen auch anwenden zu können soll das folgende Programmbeispiel betrachtet werden:

$$[x := a+b]^1; [y := a+b]^2; while [y > a+b]^3 do([a := a+1]^4; [x := a+b]^5)$$

Wir wenden zuerst die *kill* und *gen* Funktionen, die wir zuvor definiert haben, auf jeden einzelnen Block des Programmbeispieles an. Dadurch erhält man die folgende Tabelle:

l	$kill_{AE}(l)$	$gen_{AE}(l)$
1	$\{\}$	$\{a+b\}$
2	$\{\}$	$\{a*b\}$
3	$\{\}$	$\{a+b\}$
4	$\{a+b, a*b, a+1\}$	$\{\}$
5	$\{\}$	$\{a+b\}$

Aus dieser Tabelle können wir bereits einige wichtige Erkenntnisse gewinnen. Es geht hervor, dass nur im Label 4 die *kill*-Funktion eine nicht leere Menge zurückliefert, da hier die Variable a neu definiert wird. Weil aber diese Variable bereits in den Ausdrücken 1, 2 und in 4 verwendet wurde, sind diese Ausdrücke nach der Zuweisung in Block 4 für den weiteren Verlauf nicht mehr verfügbar. Desweiteren wird ersichtlich, dass in allen anderen Labels ein Ausdruck generiert, also verfügbar gemacht wird.
Mit diesen Funktionen und den daraus entwickelten Erkenntnissen stellen wir die entscheidenden Eingangs- und Ausgangsfunktionen der *Available Expression Analysis* auf:

$$\begin{aligned}
&AE_{entry}(1) = \{\}\\
&AE_{entry}(2) = AE_{exit}(1)\\
&AE_{entry}(3) = AE_{exit}(2) \cap AE_{exit}(5)\\
&AE_{entry}(4) = AE_{exit}(3)\\
&AE_{entry}(5) = AE_{exit}(4)
\end{aligned}$$

$$\begin{aligned}
&AE_{exit}(1) = AE_{entry}(1) \cup \{a+b\}\\
&AE_{exit}(2) = AE_{entry}(2) \cup \{a*b\}\\
&AE_{exit}(3) = AE_{entry}(3) \cup \{a+b\}\\
&AE_{exit}(4) = AE_{entry}(4) \backslash \{a+b, a*b, a+1\}\\
&AE_{exit}(5) = AE_{entry}(5) \cup \{a+b\}
\end{aligned}$$

Diese Funktionen gilt es nun rekursiv aufzulösen und anschließend zu vereinfachen. Dadurch erhalten wir dann die folgende Lösungstabelle:

l	$AE_{entry}(l)$	$AE_{exit}(l)$
1	$\{\}$	$\{a+b\}$
2	$\{a+b\}$	$\{a+b, a*b\}$
3	$\{a+b\}$	$\{a+b\}$
4	$\{a+b\}$	$\{\}$
5	$\{\}$	$\{a+b\}$

Die „available expressions analysis“ ist somit abgeschlossen und aus der Lösungstabelle geht hervor, dass

- die Variable a innerhalb der Schleife neu definiert wird,
- der Ausdruck $\{a+b\}$ immer am Schleifenanfang und
- $\{a*b\}$ nur beim ersten Schleifeneingang verfügbar ist, da er innerhalb der Schleife „gekilled“ wird.

Wir haben nun das gegebene Programm in Bezug auf seine verfügbaren Ausdrücke hin untersucht. Nun wollen wir uns anschauen, an welcher Stelle im Programm, welche Variablendefinition erreicht wird. Dies geschieht mit Hilfe der „Analyse der zu erreichenden Definitionen“.

2.3 Reaching Definition Analysis

Diese Art der Analyse ist der vorhergehenden sehr ähnlich. Es handelt sich hierbei ebenfalls um eine Vorwärtsanalyse mit dem Ziel den gegebenen Quellcode zu optimieren. Allerdings wird das Programmstück auf Zuweisungen der Variablen hin untersucht und nicht wie vorher auf einzelne Ausdrücke geachtet. Durch Kenntnis der jeweils erreichbaren Definitionen sollte dann immer klar sein, welche Werte eine Variable an einer bestimmten Stelle im Programm inne hat. Die Hauptfrage lautet, welche Definitionen an welcher Stelle im Programm erreichbar sind? Anders ausgedrückt bedeutet dies, dass man an einem Programmpunkt darauf achtet, an welcher Stelle die betrachtete Variable definiert wurde.
Im Beispiel: $[x := 2]^l; [y := 5]^2; [skip]^3$ bekämen wir am Ausgang von Label 2 die Menge $\{(x,1),(y,2)\}$, was bedeutet, dass x bei Label 1 und y bei Label 2 definiert wurde.

Um eine genaue Analyse durchführen zu können, müssen zunächst wieder die *kill* und *gen* Funktionen definiert werden:

$$kill_{RD}, gen_{RD} : Blocks_* \rightarrow P(Var_* \times Lab_*^?)$$

$kill_{RD}([x := a]^l) = \{(x,?)\} \cup \{(x,l')|B^{l'}\}$
$kill_{RD}([skip]^l) = \{\}$

$$kill_{RD}([b]^l) = \{\}$$

$$gen_{RD}([x := a]^l) = \{(x, l)\}$$
$$gen_{RD}([skip]^l) = \{\}$$
$$gen_{RD}([b]^l) = \{\}$$

Man erkennt, dass *kill* und *gen* auf skip und boolesche Bedingung keinen Einfluss haben, also eine leere Menge zurückliefern, aber bei einer Zuweisung die entsprechende Definitionen generieren beziehungsweise „killen“. Auffallend ist, dass die *kill*-Funktion jede mögliche Definition der Variablen x löscht, also selbst die Definition an einer unbekannten Stelle, was mit dem „?“ gekennzeichnet ist.
Nun können die Eingangs- und Ausgangsfunktionen der *Reaching Definition Analysis* aufgestellt werden:

$$RD_{entry}, RD_{exit} : Blocks_* \to P(Var_* \times Lab_*^?)$$

$$RD_{entry}(l) = \left\{ \begin{array}{c} \{(x, ?) | x \in FV(S_*)\} \quad falls \quad l = init(S_*) \\ \bigcup \{RD_{exit}(l') | (l', l) \in flow(S_*)\} \end{array} \right\}$$
$$RD_{exit}(l) = \begin{array}{c} (RD_{entry}(l) \backslash kill_{RD}(B^l)) \cup gen_{RD}(B^l) \\ wobei \quad B^l \in blocks(S_*) \end{array}$$

Diese Funktionen liefern das Label des Blocks mit der Variablen, an dem sie definiert worden ist. Wenn das Label der korrespondierenden Definitionsstelle am betrachteten Programmpunkt nicht bekannt ist, wie zum Beispiel zu Beginn des Programms, dann wird ein „?“ ausgegeben.
Man betrachte nun folgendes Beispiel und die korrespondierenden *kill* und *gen* Funktionen:

$$[x := 5]^1; [y := 1]^2; while [x > 1]^3 do([y := x * y]^4; [x := x - 1]^5)$$

l	$kill_{RD}(l)$	$gen_{RD}(l)$
1	$\{(x, ?), (x, 1), (x, 5)\}$	$\{(x, 1)\}$
2	$\{(y, ?), (y, 2), (y, 4)\}$	$\{(y, 2)\}$
3	$\{\}$	$\{\}$
4	$\{(y, ?), (y, 2), (y, 4)\}$	$\{(y, 4)\}$
5	$\{(x, ?), (x, 1), (x, 5)\}$	$\{(x, 5)\}$

Da Label 3 lediglich eine boolesche Bedingung ist, geben die *kill* und *gen* Funktionen an dieser Stelle eine leere Menge aus. An allen anderen Stellen wird eine Variable definiert und dadurch mehrere Definitionen unerreichbar gemacht.
Wir können somit wieder die Eingangs- und Ausgangsgleichungen aufstellen:

$$RD_{entry}(1) = \{(x,?),(y,?)\}$$
$$RD_{entry}(2) = RD_{exit}(1) RD_{entry}(3) = RD_{exit}(2) \cup RD_{exit}(5)$$
$$RD_{entry}(4) = RD_{exit}(3)$$
$$RD_{entry}(5) = RD_{exit}(4)$$

$$RD_{exit}(1) = (RD_{entry}(1) \backslash \{(x,?),(x,1),(x,5)\}) \cup \{(x,1)\}$$
$$RD_{exit}(2) = (RD_{entry}(2) \backslash \{(y,?),(y,2),(y,4)\}) \cup \{(y,2)\}$$
$$RD_{exit}(3) = RD_{entry}(3)$$
$$RD_{exit}(4) = (RD_{entry}(4) \backslash \{(y,?),(y,2),(y,4)\}) \cup \{(y,4)\}$$
$$RD_{exit}(5) = (RD_{entry}(5) \backslash \{(x,?),(x,1),(x,5)\}) \cup \{(x,5)\}$$

Durch rekursive Auflösung und Vereinfachung erhält man die Ergebnistabelle:

l	$RD_{entry}(l)$	$RD_{exit}(l)$
1	$\{(x,?),(y,?)\}$	$\{(y,?),(x,1)\}$
2	$\{(y,?),(x,1)\}$	$\{(x,1),(y,2)\}$
3	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$
4	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(y,4),(x,5)\}$
5	$\{(x,1),(y,4),(x,5)\}$	$\{(y,4),(x,5)\}$

Aus der Tabelle lässt sich nun genau erschließen, zu welchem Zeitpunkt eine bestimmte Definition erreichbar ist. So ist zum Beispiel am Ausgang von Label 4 die Variable y auch im selbigem Label definiert worden, wobei x an dieser Stelle seine Definition von Label 1 (beim ersten Schleifendurchgang) oder aber auch von Label 5 (bei allen anderen Schleifendurchgänge) erhalten kann.

Nach diesen beiden Arten der Vorwärtsanalyse sollen nun im Folgenden zwei Konzepte der Rückwärtsanalyse erläutert werden.

2.4 Very Busy Expressions Analysis

Um diese Analyse zu verstehen, muss zunächst mal der Begriff „very busy“ erklärt werden. Man bezeichnet einen Ausdruck als „very busy“, wenn am Ende des Labels eines betrachteten Ausdrucks sichergestellt ist, dass dieser im weiteren Verlauf benötigt wird. Dabei muss ebenfalls gewährleistet sein, dass dies auf jedem möglichen Programmpfad der Fall ist. Betrachtet man zum Beispiel einen Ausdruck der vor einer Fallunterscheidung liegt, so ist dieser nur dann „very busy“ wenn er sowohl im „then“ als auch im „else“ Teil wieder Verwendung findet und vor dieser erneuten Verwendung nicht verändert wird!
Ziel dieser Analyse ist ebenfalls eine Optimierung des Quellcodes, welche

zum Beispiel durch Vorziehen (*hoisting*) eines „very busy“-Ausdrucks, realisiert werden kann. Zum leichteren Verständnis wird folgendes Beispiel betrachtet:

$$if[a > b]^1 then([x := b - a]^2; [y := a - b]^3) else([y := b - a]^4; [x := a - b]^5)$$

An diesem Beispiel lässt sich erkennen, dass die Ausdrücke $\{a - b\}$ und $\{b - a\}$ zum Zeitpunkt der Bedingung beide „very busy“ sind. Dies bedeutet, dass man diese beiden Ausdrücke aus dem *if-then-else-Konstruk* herausziehen und vor die Bedingung setzten kann. Damit wäre dieser Code optimiert.

Um eine genaue Analyse vornehmen zu können benötigt man ebenfalls die Eingangs- und Ausgangsfunktionen der *Very Busy Expressions Analysis*. Dazu werden wieder die *kill* und *gen* Funktionen definiert:

$$kill_{VB}, gen_{VB} : Blocks_* \to P(AExp_*)$$

$$\begin{aligned}
&kill_{VB}([x := a]^l) = \{a' \in AExp_* | x \in FV(a')\} \\
&kill_{VB}([skip]^l) = \{\} \\
&kill_{VB}([b]^l) = \{\}
\end{aligned}$$

$$\begin{aligned}
&gen_{VB}([x := a]^l) = AExp(a) \\
&gen_{VB}([skip]^l) = \{\} \\
&gen_{VB}([b]^l) = AExp(b)
\end{aligned}$$

Die *kill*-Funktion liefert bei einer Zuweisung die Menge a' für die gilt, dass die Variable der Zuweisung x eine freie Variable von a' ist. Anders ausgedrückt bedeutet dies, dass ein Ausdruck nicht mehr „very busy“ ist, wenn einer seiner Variablen neu definiert wird. Die *gen*-Funktionen liefern bei einer Zuewisung und einer booleschen Bedingung einen neuen Ausdruck, der „very busy“ ist. Die Eingangs- und Ausgangsfunktionen sehen dann wie folgt aus:

$$VB_{entry}, VB_{exit} : Lab_* \to P(AExp_*)$$

$$\begin{aligned}
VB_{exit}(l) &= \left\{ \begin{array}{c} \{\} \quad falls \quad l \in final(S_*) \\ \bigcap \{VB_{entry}(l') | (l', l) \in flow^R(S_*)\} \end{array} \right\} \\
VB_{entry}(l) &= (VB_{exit}(l) \backslash kill_{VB}(B^l)) \cup gen_{VB}(B^l), B^l \in blocks(S_*)
\end{aligned}$$

Es ist zu beachten, dass die Ausgangsfunktion am Ende des letzten Labels eine leere Menge zurückgeben muss, da nach diesem Punkt das Programm zu Ende ist und somit kein Ausdruck mehr benutzt werden kann, also auch

keiner zu diesem Zeitpunkt „very busy“ ist. An allen anderen Programmstellen bekommen wir die Ausdrücke zurück, die am Eingang des nächsten Labels „very busy“ sind.
Die Eingangsfunktion liefert hingegen alle „very busy“ Ausdrücke, die am Ende des Labels existent sind, ohne diejenigen, welche innerhalb dieses Blocks „gekilled“ werden.
Wir wenden nun gleich die beiden Funktionen auf das obige Beispiel an und erhalten nach Vereinfachung der rekursiven Gleichungen die Ergebnistabelle:

l	$VB_{entry}(l)$	$VB_{exit}(l)$
1	$\{a-b, b-a\}$	$\{a-b, b-a\}$
2	$\{a-b, b-a\}$	$\{a-b\}$
3	$\{a-b\}$	$\{\}$
4	$\{a-b, b-a\}$	$\{a-b\}$
5	$\{a-b\}$	$\{\}$

Daraus geht hervor, dass am Ausgang des letzten Labels von jedem Programmpfad, also nach Label 3 und 5, natürlich kein Ausdruck mehr „very busy“ sein kann, da hier ja dann das Programm zu ende ist. An dessen Eingängen hingegen muss derjenige Ausdruck „very busy“ sein, der im jeweiligen Block noch benutzt wird, also hier in beiden Fällen $\{a-b\}$. Außerdem erhalten wir unsere vorherige Aussage bestätigt, dass jeweils $\{a-b\}$ und $\{b-a\}$ bereits vor der Bedingung „very busy“ sind.

Damit wurden alle Ausdrücke, die im Programm „very busy“ sind aufgelistet und wir kommen zu einer weiteren Rückwärtsanalyse.

2.5 Live Variables Analysis

Auch hier gilt es sich zunächst einmal die Begrifflichkeiten vor Auge zu führen und diese zu verstehen.
Eine Variable wird als „live“ bezeichnet, wenn sie am Ende des betrachteten Labels einen Wert enthält, der im weiteren Programmverlauf noch gebraucht werden könnte.
Die Informationen, die sich aus der Analyse der „lebendigen Variablen“ ergeben, können dazu verwendet werden, nicht benutzten Code zu lokalisieren, um diesen anschließend zu löschen (engl.: *dead code elimination*). Außerdem können zwei Variablen, welche nie zur gleichen Zeit „live“ sind im selben Register gespeichert werden, was wiederum Speicherplatz spart.
Um den Begriff zu veranschaulichen betrachte man das folgende Beispiel:

$$[x := 2]^1; [y := 4]^2; [x := 1]^3;$$
$$(if [y > x]^4 then [z := y]^5 else [z := y * y]^6); [x := z]^7$$

In diesem Programmcode ist die Variable x am Label 1 nicht „live“, da sie im Label 3 überschrieben und bis dahin nicht gebraucht wird. Am Ende von Label 3 ist sie dann aber „live“, da sie bereits bei der Bedingung in Label 4 benötigt wird.
Da x am Ausgang von Label 1 nicht „live“ ist, kann man die Zuweisung des Blocks mit dem Label 1 auch ohne Veränderung des Datenflusses löschen. Dies kann man auch anschaulich verdeutlichen, wenn man sich überlegt, dass auf eine Definition einer Variablen, die nie benötigt wird, auch gleich verzichtet werden kann.

Für eine vollständige Analyse sind natürlich wieder die *gen*, *kill*, Eingangs- und Ausgangsfunktionen der *Live Variables Analysis* von nöten. Sie sind folgendermaßen definiert:

$$kill_{LV}, gen_{LV} : Blocks_* \to P(Var_*)$$
$$LV_{entry}, LV_{exit} : Lab_* \to p(Var_*)$$

$kill_{LV}([x := a]^l) = \{x\}$
$kill_{LV}([skip]^l) = \{\}$
$kill_{LV}([b]^l) = \{\}$

$gen_{LV}([x := a]^l) = FV(a)$
$gen_{LV}([skip]^l) = \{\}$
$gen_{LV}([b]^l) = FV(b)$

$$LV_{exit}(l) = \left\{ \begin{array}{c} \{\} \quad falls \quad l = final(S_*) \\ \bigcup\{LV_{entry}(l') | (l', l) \in flow^R(S_*)\} \end{array} \right\}$$
$$LV_{entry}(l) = (LV_{exit}(l) \backslash kill_{LV}(B^l), B^l \in blocks(S_*)$$

Die *kill* Funktion hat auf skip und boolesche Aussagen wieder keinen Einfluss, gibt aber bei einer Zuweisung diejenige Variable zurück, welche überschrieben wird.
Die *gen* Funktion liefert bei einer Zuweisung die freie Variable von der zugewiesenen Variable a und bei einer booleschen Bedingung die freie Variable von derjenigen, die bei der booleschen Bedingung verwendet wird. Dies bedeutet, dass die *gen*-Funktion nur rückwirkend „live“ Variablen erzeugen kann, da eine definierte Variable erst dann „live“ ist, wenn sie verwendet wird. Daher ist die „Live Variables Analysis“ eine Rückwärtsanalyse.
Wie schon bei der „Very Busy Expressions Analysis“ gibt die Ausgangsfunktion am Ende des letzten Labels eine leere Menge zurück, da hier das Programm zu ende ist und somit keine Variable mehr verwendet wird, also auch keine mehr als „live“ gekennzeichnet werden kann. An allen anderen

Programmstellen liefert die Ausgangsfunktion die Menge an „live“ Variablen, die am Eingang des nächsten Blocks vorkommen.
Die Eingangsfunktion gibt die Menge der „live“ Variablen zurück, die auch am Ausgang des betrachteten Labels existieren ohne diejenigen, die innerhalb des untersuchten Blocks „gekilled“ werden.
Wendet man nun die *kill* und *gen* Funktionen am obigen Beispiel an, erhält man die Tabelle:

l	$kill_{LV}(l)$	$gen_{LV}(l)$
1	$\{x\}$	$\{\}$
2	$\{y\}$	$\{\}$
3	$\{x\}$	$\{\}$
4	$\{\}$	$\{x, y\}$
5	$\{z\}$	$\{y\}$
6	$\{z\}$	$\{y\}$
7	$\{x\}$	$\{z\}$

Es fällt auf, dass in den ersten 3 Label keine „live“ Variablen generiet werden, da hier lediglich die Variablen definiert werden. Erst wenn die definierten Variablen in den Label 4,5,6 und 7 benutzt werden, dann werden diese auch als „live“ gekennzeichnet.
Nach Aufstellen der Eingangs- und Ausgangsfunktionen und ihrer rekursiven Anwendung auf das Beispielprogramm erhalten wir wieder die Ergebnistabelle:

l	$LV_{entry}(l)$	$LV_{exit}(l)$
1	$\{\}$	$\{\}$
2	$\{\}$	$\{y\}$
3	$\{y\}$	$\{x, y\}$
4	$\{x, y\}$	$\{y\}$
5	$\{y\}$	$\{z\}$
6	$\{y\}$	$\{z\}$
7	$\{z\}$	$\{\}$

Darin enthalten sind nun alle Variablen, die zu einem bestimmten Zeitpunkt „live“ sind. Es ist anzumerken, dass das Programm am Ausgang des letzten Labels keine „live Variable“ mehr enthalten kann, also an dieser Stelle eine leere Menge in der Tabelle stehen muss. Man erkennt auch, dass die obige Aussage bekräftigt wurde, nämlich, dass die Variable x am Programmanfang nicht live ist, hingegen am Ausgang von Label 3 schon.

Im folgenden letzten Unterkapitel dieser Seminararbeit wird nun ein Konzept vorgestellt, welches auf der *Live Variable Analysis* aufbaut.

2.6 Derived Data Flow Information

Die so genannte „abgeleitete Datenflussinformation“ schafft auf der Grundlage der *Live Variables Analysis* die Verbindungen zwischen den Labels, an dem eine Variable definiert wird und den Programmstellen, an denen diese benutzt wird.
Dabei sind zwei Betrachtungsrichtungen möglich, die jeweils eine eigene Funktion besitzen:

1. „Use-Definition chains“ oder kurz „ud-chains“ geht vom jeweiligen Block aus, in welchem eine Variable benutzt wird und gibt das Label des Blocks zurück, in welchem diese Variable definiert wurde.

2. „Definition-Use chains“ oder kurz „du-chains“ geht vom jeweiligen Block aus, in welchem eine Variable definiert wird und gibt alle Labels derjenigen Blöcke zurück, in denen diese definierte Variable benutzt wird.

Es sollte nach Betrachtung des vorhergehenden Unterkapitels klar sein, dass wenn eine Variable definiert, aber vor ihrer Benutzung überschrieben wird, diese dann auch nicht „live“ ist. Dies lässt sich nun auch durch die beiden Funktionen „ud-chains“ und „du-chains“ feststellen.
Die *Derived Data Flow Information* kann damit ebenfalls zur „dead code elimination“ verwendet werden und gegebenenfalls auch für das sogennante „hoisting“, welches bereits bei der *Very Busy Expressions Analysis* erwähnt wurde.

Im Folgenden sollen nun die beiden chains-Funktionen definiert werden. Dazu benötigt man zuerts die zwei Hilfsfunktionen $use(x, l)$ und $def(x, l)$.
$use(x, l)$ überprüft, ob eine Variable x im Block mit dem Label l benutzt wird, wohingegen $def(x, l)$ untersucht, ob die Variable x an der Stelle l definiert wird:

$$use(x, l) = (\exists B : [B]^l \in blocks(S_*) \wedge x \in gen_{LV}([B]^l))$$

$$def(x, l) = (\exists B : [B]^l \in blocks(S_*) \wedge x \in kill_{LV}([B]^l))$$

Als weitere Hilfsfunktion wird $clear(x, l, l')$ definiert. Diese soll einen klaren Definitionspfad beschreiben und sieht wie folgt aus:

$$clear(x, l, l') = \exists l_1, ..., l_n :$$

$$\begin{gathered}(l_1 = l) \wedge (l_n = l') \wedge (n > 0) \wedge \\ (\forall i \in \{1, ..., n-1\} : (l_i, l_{i+1}) \in flow(S_*)) \wedge \\ (\forall i \in \{1, ..., n-1\} : \neg def(x, l_i)) \wedge use(x, l_n)\end{gathered}$$

Die Label $l_1, ...l_n$ ergeben einen klaren Definitionspfad für die Variable x, wenn keiner der Blöcke mit den Label $l_1, ...l_n$ die Variable x definiert und wenn der Block mit dem Label l_n die Variable x benutzt.

Mit Hilfe dieser Definitionen werden nun die beiden „chains“-Funktionen beschrieben:

$$ud, du : Var_* \times Lab_* \rightarrow P(Lab_*)$$

$$\begin{aligned} ud(x, l') = &\{l | def(x, l) \wedge \exists l'' : (l, l'') \in flow(S_*) \wedge clear(x, l'', l'\} \\ &\cup \{?|clear(x, init(S_*), l')\} \end{aligned}$$

$$du(x, l) = \left\{ \begin{array}{c} \{l'|def(x, l) \wedge \exists l'' : (l, l'') \in flow(S_*) \wedge \\ clear(x, l'', l')\} \quad falls \quad l \neq ? \\ \{l'|clear(x, init(S_*), l')\} \quad falls \quad l = ? \end{array} \right\}$$

Wie aus der formalen Beschreibung hervorgeht, gibt $ud(x, l')$ das Label zurück, an dem x - nachweislich durch $def(x, l)$ - definiert wurde oder es wird durch ein ? gezeigt, dass x noch nicht initialisiert ist.
Für $du(x, l)$ werden die Labels ausgegeben, an denen x, was an der Stelle l definiert wurde, benutzt wird. Hierbei muss auch unterschieden werden, ob x innerhalb des betrachteten Programms initialisiert oder undefiniert ist, was ebenfalls mit einem „?“ signalisiert wird.

Um nun die Theorien dieser Analyse etwas mehr zu veranschaulichen, betrachte man folgendes Beispiel und die darauf angewendeten $ud - chains$ beziehungsweise $du - chains$ Funktionen in den darunterliegenden Tabellen:

$$[x := 0]^1; [x := 3]^2; (if [z = x]^3 then [z := 0]^4 else [z := x]^5);$$
$$[y := x]^6; [x := y + z]^7$$

ud(x,l)	x	y	z
1	{}	{}	{}
2	{}	{}	{}
3	{2}	{}	{?}
4	{}	{}	{}
5	{2}	{}	{}
6	{2}	{}	{}
7	{}	{6}	{4,5}

du(x,l)	x	y	z
1	{}	{}	{}
2	$\{3,5,6\}$	{}	{}
3	{}	{}	{}
4	{}	{}	$\{7\}$
5	{}	{}	$\{7\}$
6	{}	$\{7\}$	{}
7	{}	{}	{}
?	{}	{}	$\{3\}$

Betrachtet man die erste Tabelle, lässt sich sehr leicht die beschriebene „Use-Definition“ nachvollziehen. Beispielsweise kann man ablesen, dass die Variable x, welche in Label 5 benutzt wird, im Block mit dem Label 2 definiert wurde.
In der zweiten Tabelle wird beispielsweise die Information gegeben, dass die Variable x, die im Block mit dem Label 2 definiert ist, bei den Label 3, 5 und 6 benutzt wird.
Sieht man sich nun beide Tabellen gleichzeitig an, so fällt auf, dass jeweils in der ersten Zeile nur leere Mengen vorkommen. Das kommt daher, dass die Variable x, welche im Block mit dem Label 1 definiert ist, keine „live Variable“ ist und somit nicht benutzt wird. Folglich lässt sich der erste Block des Programms auch eliminieren und wir haben wieder eine Optimierung des Quellcodes vorgenommen.
Ebenso geht aus den beiden Tabellen hervor, dass der Block mit dem Label 6 auch vor die Bedingung gezogen werden kann, da die Variable y, die bei Label 6 definiert wird, im if-then-else Konstrukt nicht benötigt wird und gleichzeitig keine Definitionen der Labels 3,4 oder 5 benutzt. Diesen Vorgang nennt man auch „Code Motion“ oder wie schon beschrieben „hoisting“.

Es sollte nun klar geworden sein, wie man an eine intraprozedurale Analyse herangeht und welchen Nutzen man davon hat. Außerdem dürfte nun die Zuordnung von Problemstellung und der dafür geeigneten Analyseart keine Schwierigkeiten mehr darstellen.

Im letzten Kapitel dieser Seminararbeit geht es nun um die Semantik und die theoretischen Eigenschaften der vorgestellten Analysen. Im Speziellen wird dabei die Richtigkeit der „Live variable analysis“ gezeigt, so wie sie auch von den Autoren der Begleitlektüre erläutert wird.

3 Theoretische Eigenschaften

Um die Richtigkeit der *Live Variables Analysis* beweisen zu können, bedient man sich einer strukturellen Semantik, die es gestattet Programme zu untersuchen, welche nicht terminieren.

Als erstes wird ein Zustand σ definiert, der eine Variable Var auf einen Integer Z abbildet:

$$\sigma \in State = Var \rightarrow Z$$

Als nächstes beschreiben wir die *Transition der Semantik*:

$$\text{I, } \langle S, \sigma \rangle \rightarrow \sigma'$$
$$\text{II, } \langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$$

Es wird ersichtlich, dass die Konfiguration $\langle S, \sigma \rangle$ nur 2 Fälle nach sich ziehen kann:

1. Die Ausführung terminiert nach einem Schritt und somit ergibt sich die Funktion I, oder
2. die Ausführung terminiert nicht nach diesem Schritt und man erhält eine neue Konfiguration $\langle S', \sigma' \rangle$ nach Funktion II.

Mit diesen Definitionen wird nun im anschließenden Unterkapitel die *Live Variables Analysis* auf ihre Korrektheit überprüft.

3.1 Die Richtigkeit der Liver Variables Analysis

Da die theoretischen Annahmen und Beweise, die einen eindeutigen Schluss auf die Richtigkeit der *Live Variables Analysis* zulassen, sehr komplex sind, soll in dieser Seminararbeit lediglich das entscheidende Theorem erklärt und an einem Beispiel veranschaulicht werden.
Man rufe sich zunächst die Eingangs- und Ausgangsfunktionen der *Live Variables Analysis* ins Gedächtnis und betrachte davon eine globale Funktion *live*.
Damit können wir definieren:

$live| = LV^{=}(S)$ man spricht: *live* löst $LV^{=}(S)$
$live| = LV^{\subseteq}(S)$ man sagt: *live* löst $LV^{\subseteq}(S)$

Wenn nun $live| = LV^{\subseteq}(S)$ und S ist *Label konsistent* dann gilt das folgende Theorem:

I, falls $\langle S, \sigma_1 \rangle \rightarrow \langle S', \sigma_1' \rangle$ und $\sigma_1 \cong_{N(init(S))} \sigma_2$ dann existiert auch ein σ_2, so dass $\langle S, \sigma_2 \rangle \rightarrow \langle S', \sigma_2' \rangle$ und $\sigma_1 \cong_{N(init(S'))} \sigma_2'$, und

II, falls $\langle S, \sigma_1 \rangle \rightarrow \sigma_1'$ und $\sigma_1 \cong_{N(init(S))} \sigma_2$ dann existiert auch ein σ_2' so dass $\langle S, \sigma_2 \rangle \rightarrow \sigma_2'$ und $\sigma_1' \cong_{X(init(S))} \sigma_2'$

Mit $\sigma_1 \cong_V \sigma_2$ falls $\forall x \in V : \sigma_1(x) = \sigma_2(x)$

In Worten ausgedrückt bedeutet dieses Theorem, dass:

1. wenn wir ein Programm haben, das nicht nach einem Schritt terminiert und $\sigma_1(x) = \sigma_2(x) \forall x \in N(init(S))$, dann existiert auch eine zweite Konfiguration mit einem σ_2' und es gilt $\sigma_1'(x) = \sigma_2'(x) \forall x \in N(init(S'))$. $N(init(S))$ gibt dabei die Menge der Variablen wieder, die am Anfang von Programm S beziehungsweise am Anfang vom Rest des Programms S' „live“ sind.

2. wenn wir ein Programm haben, das nach einem Schritt terminiert und $\sigma_1(x) = \sigma_2(x) \forall x \in N(init(S))$, dann existiert auch eine zweite Konfiguration mit einem σ_2' und es gilt $\sigma_1'(x) = \sigma_2'(x) \forall x \in X(init(S))$. $X(init(S))$ gibt dabei die Menge der Variablen wieder, die am Ausgang des ersten Blocks „live“ sind.

Veranschaulichen wir dies nun mit einem geeigneten Beispiel:

$$\left\langle [x := y + z]^l, \sigma_1 \right\rangle$$

Bei diesem einfachen Beispiel handelt es sich um den letzten Programmschritt eines größeren Programmcodes. Man sieht eine Zuweisung, welche der Variablen x die Summme der Werte der Variablen y und z zuweist. Die Werte der Variablen sind dabei stets in σ gespeichert. Desweiteren soll es ein V_1 geben, dass die zuweisenden Variabelen beinhaltet, also y und z und es wird ein V_2 definiert, welches die Variable x in ihrer Menge enthält. Mathematisch ausgedrückt heißt das:

$$V_1 = y, z \text{ und}$$
$$V_2 = x$$

Nun lassen sich V_1 und V_2 dazu benutzen, um folgende Zusammenhänge zu definieren:

$$\sigma_1 \cong_{V_1} \sigma_2 <=> \sigma_1(y) = \sigma_2(y) \wedge \sigma_1(z) = \sigma_2(z)$$
$$\sigma_1 \cong_{V_2} \sigma_2 <=> \sigma_1(x) = \sigma_2(x)$$

Daraus geht nun hervor, dass σ_1 auf V_1 und V_2 angewendet das gleiche ist, wie σ_2.
Führen wir nun den letzten Programmschritt bezüglich σ_1 und σ_2 an unserem Beispiel durch, erhalten wir:

$$\left\langle [x := y + z]^l, \sigma_1 \right\rangle \rightarrow \sigma_1' \wedge \left\langle [x := y + z]^l, \sigma_2 \right\rangle \rightarrow \sigma_2'$$

Wenn man sich das oben beschriebene Theorem betrachtet, dann stellt man fest, dass es sich bei dieser Notation der letzten Programmausführung, genau um den Fall *II* des Theorems handelt. Es geht also unmittelbar daraus hervor, dass wenn $\sigma_1 \cong_{V_1} \sigma_2$ gilt, folglich auch ein σ_2' existieren muss, so dass: $\sigma_1' \cong_{V_2} \sigma_2'$ gilt. Aus dieser Schlussfolgerung ergibt sich nun der Beweis dafür, dass V_2 am Ende des Blocks existiert, also „live“ ist und V_2 unmittelbar von V_1 abhängt. Anschaulich darstellbar ist diese Abhängigkeit auch allein durch die Betrachtung der Zuweisung im gegebenen Beispiel. Da x die Summe der Werte von y und z erhält, muss folglich auch x von y und z abhängig sein. Es ist also zu erkennen, dass sich mit Hilfe des Theorems die Richtigkeit der *Live Variables Analysis* zeigen lässt.

4 Zusammenfassung

Es wurden nun in dieser Seminararbeit fünf Formen der intraprozeduralen Analyse vorgestellt und erklärt. Neben diesen gibt es noch weitere Analysearten, die aber weder hier noch in der zugrunde liegenden Literatur beschrieben werden. Wichtig ist zu wissen, dass die Datenflussanalyse die Grundlage bildet, um einen bereits geschriebenen Code zu optimieren und zu vereinfachen. Dabei geht man wie folgt vor:

- Gegebenes Problem und dazugehörigen Code betrachten
- Analyseart wählen
- Die korresponiderenden Gleichungen aufstellen
- Auf den gegebenen Code anweden und vereinfachen
- Ergebnis auswerten
- Code optimieren und eventuelle Probleme lösen.

Um die richtige Analyseart zu wählen, muss man sich fragen, was genau untersucht werden soll und wählt dann zwischen *Available Expressions Analysis*, *Reaching Definitions Analysis*, *Very Busy Expressions Analysis*, *Live Variables Analysis* oder *Derived Data Flow Information* aus.

Literatur

Principles of Program Analysis
Flemming Nielson, Hanne Riis Nielson, Chris Hankin
(Springer-Verlag, 2005)