Ludwig-Maximilians-Universität München
Department „Institut für Informatik“
Lehr- und Forschungseinheit für Theoretische Informatik
Prof. Martin Hofmann, Ph.D.

Haupt-/Bachelorseminar: Programmanalyse

1. Einführung

Benedikt Zierer
zierer@cip.ifi.lmu.de

Bearbeitungszeitraum: 21. 4. 2009 bis 8. 7. 2009
Betreuer: Dr. Martin Lange
Verantw. Hochschullehrer: Prof. Martin Hofmann, Ph.D.

Zusammenfassung

Das Ziel dieser Arbeit ist es, einen Überblick über die verschiedenen Ansätze zur Programmanalyse zu schaffen und an Beispielen zu erläutern, ohne allzu tief in die technischen Details einzusteigen, als Grundlange dient das Buch „Principles of Program Analysis“[1].

Abstract

This work targets at giving an introduction about different types of Program Analysis and to explain them with examples, without explaining the technical details in depth. This work is based on the book “Principles of Program Analysis”[1].

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, 9. Juli 2009

. . . . . .

Inhaltsverzeichnis

1 Einleitung

Die Programmanalyse wird verwendet, um zur *Kompilierzeit* Werte von Variablen und das dynamische Verhalten eines Programmes zur *Laufzeit* abzuschätzen; das Programm wird also nicht tatsächlich ausgeführt, deshalb spricht man auch von einer *statischen Analyse*.
Die Haupteinsatzzwecke einer Programmanalyse sind *redundante* und überflüssige Rechenschritte zu vermeiden (also den Compiler zu optimieren) sowie das Verhalten von Software zu *validieren*; das ist besonders wichtig wenn ein gekauftes, also nicht selbst entwickeltes, Programm im Produktivbetrieb eingesetzt werden soll.
Das Ziel dieser Arbeit ist es, einen Überblick über die verschiedenen Ansätze zur Programmanalyse zu schaffen und an Beispielen zu erläutern, ohne allzu tief in die technischen Details einzusteigen; als Grundlange dient das Buch „Principles of Program Analysis“[1], alle Textpassagen oder Abbildungen die nicht ausdrücklich aus einer anderen Quelle stammen, basieren auf dem Buch.
Insbesondere werden auf folgende vier Hauptansätze zur Programmanalyse eingegangen: 2.2 Datenflussanalyse, 2.3 Grenzwertbasierte Analyse, 2.4 Abstrakte Interpretation sowie 2.5 Typ- und Effektsysteme, darauf folgt jeweils eine kurze Einführung in 2.6 Algorithmen und 2.7 Transformationen.

2 Hauptteil

Der wichtigste Punkt aller hier vorgestellten Ansätze zur Programmanalyse ist die Tatsache, dass sich aus dem Satz von Rice
„Jede nichttriviale Eigenschaft einer Turingmaschine, die sich auf die von der TM berechnete Funktion bezieht, ist unentscheidbar.“, siehe [2].
ergibt, dass keine exakten Vorhersagen über das Verhalten von Programmen (beziehungsweise nichttrivialen Programmteilen), sondern höchstens *Abschätzungen* getroffen werden können, da das Problem (auf eine Turingmaschine bezogen) an sich unentscheidbar ist.
Um ungewolltes Verhalten eines Programmes erkennen zu können, sollte diese Abschätzung **mindestens** alle Möglichkeiten, wie sich das Programm zur Laufzeit verhalten könnte beinhalten, eine *Überapproximation* beziehungsweise ein einseitiger Irrtum ist also wünschenswert, da eine Unterapproximation möglicherweise ungewollte Ergebnisse nicht abdecken könnte, hierzu ein kleines Beispiel:
Gehen wir von einer simplen imperativen Programmiersprache aus, deren Syntax nicht weiter spezifiziert und selbsterklärend sein sollte, und stellen wir uns folgendes Programm vor:

```
read(x); (if x>0 then y:=1 else (y:=2;S)); z:=y
```

S soll hier ein Statement sein, das `y` keinen anderen Wert mehr zuweist. Intuitiv sollte man davon ausgehen, dass `z := y` zur Folge hat, dass `z` nur die Werte 1 und 2 annehmen kann, da es für `y` keine anderen Möglichkeiten gibt.
Würde nun eine Programmanalyse zu dem Ergebnis kommen, dass `z` nur den Wert 1 annehmen kann, wäre das zwar für den Fall dass $\texttt{x} \leq 0$ oder dass S mit $\texttt{x} > 0$ und $\texttt{y} = 2$ nicht terminiert richtig. Da aber nicht bekannt ist, ob und wenn ja für welche Werte S terminiert, sollte die Analyse nicht davon ausgehen, dass das unbekannte S nicht terminiert.
Aber eine Analyse, die zu dem Ergebnis kommt, dass `z` 1, 2 oder 42 annehmen kann, wäre akzeptabel, da auf jeden Fall einige Aussagen über das Ergebnis getroffen werden können, wie „das Ergebnis ist positiv“, „das Ergebnis ist eine natürliche Zahl“ oder auch „das Ergebnis lässt sich in einem Byte abspeichern“. Natürlich ist eine Analyse, die nur genau alle richtigen Ergebnisse beinhaltet, noch besser.

Außerdem sollte jede Programmanalyse auf einer Semantik basieren, so dass die Richtigkeit der Ergebnisse bewiesen werden könnte.

2.1 Verwendete Syntax

In dieser Arbeit soll die gleiche Syntax wie im Buch von Nielson [1, Seite 3-5] verwendet werden, um die Arbeitsweise von Programmanalysen zu veranschaulichen.
Dort wird eine einfache, imperative Sprache namens WHILE vorgestellt, in der ein Programm aus einem *Statement* besteht, das in der Regel aus mehreren Statements besteht. Jedes Statement erhält ein eindeutiges Label l um den Datenfluss leichter darstellen zu können, jede von Klammern umschlossene und mit einem Label versehene Anweisung wird *Block* genannt.
WHILE verfügt über folgende Syntax:

a	$\in$	**AExp**	Arithmetische Ausdrücke
b	$\in$	**BExp**	Boole'sche Ausdrücke
S	$\in$	**Stmt**	Statements
x, y	$\in$	**Var**	Variablen
n	$\in$	**Num**	Ziffern
l	$\in$	**Lab**	Label
op_a	$\in$	$\mathbf{Op}_a$	Rechenzeichen
op_b	$\in$	$\mathbf{Op}_b$	Boole'sche Operatoren
op_r	$\in$	$\mathbf{Op}_r$	Vergleichsoperatoren

$$
\begin{array}{lll}
a & ::= & x \mid n \mid a_1\ op_a\ a_2 \\
b & ::= & \textbf{true} \mid \textbf{false} \mid \textbf{not}\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_a\ a_2 \\
S & ::= & [x := a]^l \mid [\texttt{skip}]^l \mid S_1\ S_2 \mid \texttt{if}\ [b]^l\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ [b]^l\ \texttt{do}\ S
\end{array}
$$

Man kann sich diese Syntax so vorstellen, dass sie einen abstrakten Syntaxbaum aufspannt, erst die eigentliche Programmiersprache liefert genügend Informationen um hieraus einen konkreten Syntaxbaum (Parsebaum) zu machen.
Ein Beispiel in WHILE wäre folgendes Programm, das die Fakultät von `x` berechnet und in `z` speichert:

$$[\texttt{y} := \texttt{x}]^1;\ [\texttt{z} := 1]^2;\ \texttt{while}\ [\texttt{y} > 1]^3\ \texttt{do}\ ([\texttt{z} := \texttt{z} * \texttt{y}]^4;\ [\texttt{y} := \texttt{y} - 1]^5);\ [\texttt{y} := 0]^6$$

Durch die Verwendung von eindeutigen Labels können einzelne Programmteile schon ohne die direkte Erstellung eines Datenfluss-Graphen identifiziert werden, außerdem kann so eine einfache Programmanalyse durchgeführt werden, welche im Verlauf dieser Arbeit zur Erklärung der Arbeitsweisen verschiedener Ansätze verwendet wird, die Zuweisungsanalyse (Reaching Definitions Analyses):

2.1.1 Zuweisungsanalyse

Eine Zuweisung der Gestalt $[x := a]^l$ bedeutet, dass eine Möglichkeit in der Programmausführung darin besteht, dass an der Stelle l der Variable x der Wert a zugewiesen wird, diese Zuweisung kann dann von einem anderen Block verwendet werden.
Am Beispiel des Programmes zur Fakultätsberechnung

$$[\texttt{y} := \texttt{x}]^1;\ [\texttt{z} := 1]^2;\ \texttt{while}\ [\texttt{y} > 1]^3\ \texttt{do}\ ([\texttt{z} := \texttt{z} * \texttt{y}]^4;\ [\texttt{y} := \texttt{y} - 1]^5);\ [\texttt{y} := 0]^6$$

lässt sich die Vorgehensweise, festzustellen welche Variablen mit welchem Label l den Block betreten und verlassen:
$[\texttt{y} := \texttt{x}]^1$ betritt $[\texttt{z} := 1]^2$, nun werden Paare aus Variablen und den Labels, wo eine Zuweisung stattfindet, gebildet: Also erreichen hier `(y,1)` und `(x,?)` das Label 2, wobei „?“ ein spezielles Label ist, das im Programm selbst nicht vorkommen darf, welches kennzeichnet dass die zugehörige Variable (hier `x`) möglicherweise noch nicht initialisiert wurde.
Alle Paare aus Variablen und Labels lassen sich für das Betreten $\mathrm{RD}_{entry}(l)$ und Verlassen $\mathrm{RD}_{exit}(l)$ jedes Blocks l in einer Tabelle darstellen:

l	$\mathrm{RD}_{entry}(l)$	$\mathrm{RD}_{exit}(l)$
1	`(x,?),(y,?),(z,?)`	`(x,?),(y,1),(z,?)`
2	`(x,?),(y,1),(z,?)`	`(x,?),(y,1),(z,2)`
3	`(x,?),(y,1),(y,5),(z,2),(z,4)`	`(x,?),(y,1),(y,5),(z,2),(z,4)`
4	`(x,?),(y,1),(y,5),(z,2),(z,4)`	`(x,?),(y,1),(y,5),(z,4)`
5	`(x,?),(y,1),(y,5),(z,4)`	`(x,?),(y,5),(z,4)`
6	`(x,?),(y,1),(y,5),(z,2),(z,4)`	`(x,?),(y,6),(z,2),(z,4)`

Bei genauer Betrachtung wird ersichtlich, dass Blöcke der Gestalt $[b]^l$ dieselben Werte beim Betreten und Verlassen aufweisen, wohingegen sich bei Blöcken wie $[x := a]^l$ die Wertepaare (x, l') unterscheiden können: Hier kann eine neue oder alternative Zuweisung erfolgen, die im Folgenden berücksichtigt werden muss.

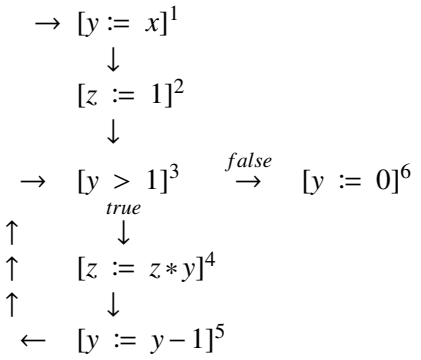
An dieser Stelle kann noch einmal der Unterschied zwischen Unter- und Überapproximation aufgezeigt werden:

Für $\mathrm{RD}_{entry}(5)$ und $\mathrm{RD}_{exit}(5)$ könnte ohne weiteres das Paar $(z, 2)$ eingefügt würden, um auszudrücken, dass z schon einmal ein Wert zugewiesen wurde, da Label 5 (in diesem Programm) aber nur auf Label 4 folgen kann, wäre dies eine weitere Überapproximation, da z in Label 4 auf jeden Fall ein Wert zugewiesen wird.

Würde man hingegen in $\mathrm{RD}_{entry}(6)$ und $\mathrm{RD}_{exit}(6)$ das Paar $(z, 2)$ entfernen, entstünde hier eine Unterapproximation, da für den Fall $x \leq 1$ kein Wert für z vorhanden wäre.

2.2 Datenflussanalyse

Die Datenflussanalyse kommt, so [2], vor allem in Compilern zum Einsatz und ist prinzipiell für imperative Programmiersprachen gedacht; sie kann auch für funktionale oder objektorientierte Sprachen verwendet werden, solange sich ein Datenfluss-Graph zeichnen lässt.
Die Analyse basiert darauf, alle Möglichkeiten eines Flussdiagramms zu durchlaufen.
Der Graph besteht aus (durch das Label l eindeutig gekennzeichneten) Blöcken als Knoten und gerichteten Kanten an der Stellen, wo die Kontrolle von einem Block auf einen anderen übergeht. Wenn man nun das Beispielprogramm von Seite 4 als Datenfluss-Graph zeichnet, ergibt sich folgendes Bild:



2.2.1 Ansatz mit Gleichungssystem

Aus dem Programm zur Fakultätsberechnung

$$[y := x]^1 \; [z := 1]^2 \; \texttt{while}\, [y > 1]^3 \, \texttt{do}\, ([z := z*y]^4 \; [y := y-1]^5) \; [y := 0]^6$$

lassen sich folgende Gleichungen extrahieren:

$$\begin{aligned}
\mathrm{RD}_{exit}(1) &= (\mathrm{RD}_{entry}(1) \backslash \{(\mathtt{y}, l) \mid l \in \mathbf{Lab}\}) \cup \{(\mathtt{y}, 1)\} \\
\mathrm{RD}_{exit}(2) &= (\mathrm{RD}_{entry}(2) \backslash \{(\mathtt{z}, l) \mid l \in \mathbf{Lab}\}) \cup \{(\mathtt{z}, 2)\} \\
\mathrm{RD}_{exit}(3) &= \mathrm{RD}_{entry}(3) \\
\mathrm{RD}_{exit}(4) &= (\mathrm{RD}_{entry}(4) \backslash \{(\mathtt{z}, l) \mid l \in \mathbf{Lab}\}) \cup \{(\mathtt{z}, 4)\} \\
\mathrm{RD}_{exit}(5) &= (\mathrm{RD}_{entry}(5) \backslash \{(\mathtt{y}, l) \mid l \in \mathbf{Lab}\}) \cup \{(\mathtt{y}, 5)\} \\
\mathrm{RD}_{exit}(6) &= (\mathrm{RD}_{entry}(6) \backslash \{(\mathtt{y}, l) \mid l \in \mathbf{Lab}\}) \cup \{(\mathtt{y}, 6)\}
\end{aligned}$$

Alle diese Gleichungen folgen dem gleichen Schema: Bei einer Zuweisung $[x := a]^{l'}$ werden alle Paare (x, l) aus $\mathrm{RD}_{entry}(l')$ durch (x, l') ersetzt um $\mathrm{RD}_{exit}(l')$ zu erhalten - hiermit wird ausgedrückt, dass x als l neu definiert wird. Für alle anderen Blöcke $[\ldots]^{l'}$ wird $\mathrm{RD}_{exit}(l')$ gleich $\mathrm{RD}_{entry}(l')$ gesetzt um auszudrücken, dass sich keine Variablen ändern.

Eine weitere Klasse von Gleichungen verbindet Eintrittspunkte in Blöcken mit den Ausgangspunkten, von denen der Kontrollfluss aus übergegangen sein könnte. Für das Beispielprogramm erhält man folgende Gleichungen:

$$\begin{aligned}
\mathrm{RD}_{entry}(2) &= \mathrm{RD}_{exit}(1) \\
\mathrm{RD}_{entry}(3) &= \mathrm{RD}_{exit}(2) \cup \mathrm{RD}_{exit}(5) \\
\mathrm{RD}_{entry}(4) &= \mathrm{RD}_{exit}(3) \\
\mathrm{RD}_{entry}(5) &= \mathrm{RD}_{exit}(4) \\
\mathrm{RD}_{entry}(6) &= \mathrm{RD}_{exit}(3)
\end{aligned}$$

Generell schreibt man also $\mathrm{RD}_{entry}(l) = \mathrm{RD}_{exit}(l_1) \cup \ldots \cup \mathrm{RD}_{exit}(l_n)$ wenn $l_1, \ldots, l_n$ die Label sind, von denen aus der Kontrollfluss auf l übergehen könnte.
Betrachtet man nun folgende Gleichung

$$\mathrm{RD}_{entry}(l) = \{(x, ?) \mid x \text{ ist eine Variable im Programm}\}$$

wird deutlich, dass „?“ für Variablen verwendet wird, die nicht initialisiert wurden, also kann man über das Beispielprogramm folgende Aussage treffen:

$$\mathrm{RD}_{entry}(l) = \{(\mathtt{x}, ?), (\mathtt{y}, ?), (\mathtt{z}, ?)\}$$

Im Gleichungssystem von oben werden insgesamt 12 Punkte $\mathrm{RD}_{entry}(1)$,...,$\mathrm{RD}_{exit}(6)$ definiert. Schreibt man nun $\overrightarrow{\mathrm{RD}}$ für dieses Zwölftupel von Punkten, lässt sich das Gleichungssystem als eine Funktion F darstellen:

$$\overrightarrow{\mathrm{RD}} = F(\overrightarrow{\mathrm{RD}})$$

Genauer lässt sich

$$\overrightarrow{\mathrm{RD}} = (F_{entry}(1)(\overrightarrow{\mathrm{RD}}), F_{exit}(1)(\overrightarrow{\mathrm{RD}}), \ldots, F_{entry}(6)(\overrightarrow{\mathrm{RD}}), F_{exit}(6)(\overrightarrow{\mathrm{RD}}))$$

schreiben, wobei zum Beispiel gilt:

$$F_{entry}(3)(\ldots, \mathrm{RD}_{exit}(2), \ldots, \mathrm{RD}_{exit}(5), \ldots) = \mathrm{RD}_{exit}(2) \cup \mathrm{RD}_{exit}(5)$$

F wirkt also über Zwölftupeln von Mengen von Paaren aus Variablen und Labels, das kann man so ausdrücken:

$$F : (P(\mathbf{Var}_* \times \mathbf{Lab}_*))^{12} \to (P(\mathbf{Var}_* \times \mathbf{Lab}_*))^{12}$$

Selbstverständlich könnte man statt $\mathbf{Var}_*$ **Var** schreiben, allerdings erleichtert es die Darstellung ungemein, wenn man $\mathbf{Var}_*$ als endliche Teilmenge von **Var**, allen Variablen im Programm, definiert, und bei $\mathbf{Lab}_*$ genauso verfährt. Im Beispielprogramm gilt also $\mathbf{Var}_* = \{\texttt{x},\texttt{y},\texttt{z}\}$ und $\mathbf{Var}_* = \{1, \ldots, 6, ?\}$.
$(P(\mathbf{Var}_* \times \mathbf{Lab}_*))^{12}$ lässt sich teilweise ordnen, indem man

$$\overrightarrow{\mathrm{RD}} \sqsubseteq \overrightarrow{\mathrm{RD}'} \iff \forall_i : \mathrm{RD}_i \subseteq \mathrm{RD}'_i$$

setzt, wo wieder $\overrightarrow{\mathrm{RD}} = (\mathrm{RD}_1, \ldots, \mathrm{RD}_{12})$ und folglich $\overrightarrow{\mathrm{RD}'} = (\mathrm{RD}'_1, \ldots, \mathrm{RD}'_{12})$ gilt. Hiermit wird aus $(P(\mathbf{Var}_* \times \mathbf{Lab}_*))^{12}$ ein Verband, dessen kleinstes Element

$$\overrightarrow{\emptyset} = (\emptyset, \ldots, \emptyset)$$

ist und dessen obere Grenzen durch

$$\overrightarrow{\mathrm{RD}} \sqcup \overrightarrow{\mathrm{RD}'} = (\mathrm{RD}_1 \cup \mathrm{RD}'_1, \ldots, \mathrm{RD}_{12} \cup \mathrm{RD}'_{12})$$

gegeben sind. F ist eine monotone Funktion (der Beweis hierfür würde zu weit führen), das bedeutet:

$$\overrightarrow{\mathrm{RD}} \sqsubseteq \overrightarrow{\mathrm{RD}'} \text{ impliziert, dass } F(\overrightarrow{\mathrm{RD}}) \sqsubseteq F(\overrightarrow{\mathrm{RD}'})$$

Das hat zur Folge, dass Rechnungen der Art

$$\mathrm{RD}_{exit}(2) \subseteq \mathrm{RD}'_{exit}(2) \text{ und } \mathrm{RD}_{exit}(5) \subseteq \mathrm{RD}'_{exit}(5)$$

implizieren, dass gilt:

$$\mathrm{RD}_{exit}(2) \cup \mathrm{RD}_{exit}(5) \subseteq \mathrm{RD}'_{exit}(2) \cup \mathrm{RD}'_{exit}(5)$$

Betrachtet man jetzt $(F^n(\emptyset))_n$ und behält im Auge, dass $\overrightarrow{\emptyset} \sqsubseteq F(\overrightarrow{\emptyset})$ gilt sowie dass F eine monotone Funktion ist, ergibt eine Induktion nach n $F^n(\overrightarrow{\emptyset}) \sqsubseteq F^{n+1}(\overrightarrow{\emptyset})$.
Alle Elemente dieser Folge sind in $(P(\mathbf{Var}_* \times \mathbf{Lab}_*))^{12}$ enthalten, und da dies eine endliche Menge ist können nicht alle Elemente der Menge verschieden sein, so dass ein n existieren muss, für das

gilt:

$$F^{n+1}(\overrightarrow{\emptyset}) = F^n(\overrightarrow{\emptyset})$$

Da aber $F^{n+1}(\overrightarrow{\emptyset}) = F(F^n(\overrightarrow{\emptyset}))$ gilt, bedeutet das, dass $F^n(\overrightarrow{\emptyset})$ ein Fixpunkt in F ist und folglich $F^n(\overrightarrow{\emptyset})$ eine Lösung des obigen Gleichungssystems ist.

2.2.2 Ansatz mit Grenzwerten

Als Alternative zum Ansatz mit Gleichungssystemen auf Seite 7 lässt sich der Ansatz mit Grenzwerten verwenden, die Idee hierbei ist es, den Datenfluss nicht durch Gleichungen, sondern durch Ungleichungen oder Einschränkungen (Constraints) darzustellen.
Aus dem Programm zur Fakultätsberechnung

$$[\mathtt{y} := \mathtt{x}]^1; [\mathtt{z} := 1]^2; \mathtt{while}\,[\mathtt{y} > 1]^3\,\mathtt{do}\,([\mathtt{z} := \mathtt{z} * \mathtt{y}]^4; [\mathtt{y} := \mathtt{y} - 1]^5); [\mathtt{y} := 0]^6$$

lassen sich folgende Grenzwerte extrahieren:

$\mathrm{RD}_{exit}(1) \supseteq \mathrm{RD}_{entry}(1) \backslash \{(\mathtt{y}, l) \mid l \in \mathbf{Lab}\}$
$\mathrm{RD}_{exit}(1) \supseteq \{(\mathtt{y}, 1)\}$
$\mathrm{RD}_{exit}(2) \supseteq \mathrm{RD}_{entry}(2) \backslash \{(\mathtt{z}, l) \mid l \in \mathbf{Lab}\}$
$\mathrm{RD}_{exit}(2) \supseteq \{(\mathtt{z}, 2)\}$
$\mathrm{RD}_{exit}(3) \supseteq \mathrm{RD}_{entry}(3)$
$\mathrm{RD}_{exit}(4) \supseteq \mathrm{RD}_{entry}(4) \backslash \{(\mathtt{z}, l) \mid l \in \mathbf{Lab}\}$
$\mathrm{RD}_{exit}(4) \supseteq \{(\mathtt{z}, 4)\}$
$\mathrm{RD}_{exit}(5) \supseteq \mathrm{RD}_{entry}(5) \backslash \{(\mathtt{y}, l) \mid l \in \mathbf{Lab}\}$
$\mathrm{RD}_{exit}(5) \supseteq \{(\mathtt{y}, 5)\}$
$\mathrm{RD}_{exit}(6) \supseteq \mathrm{RD}_{entry}(6) \backslash \{(\mathtt{y}, l) \mid l \in \mathbf{Lab}\}$
$\mathrm{RD}_{exit}(6) \supseteq \{(\mathtt{y}, 6)\}$

Das obige System folgt einem Schema: Bei einer Zuweisung $[x := a]^{l'}$ existiert eine Grenze, die alle Paare (x, l) aus $\mathrm{RD}_{entry}(l')$ daran hindert, $\mathrm{RD}_{exit}(l')$ zu erreichen und für alle anderen Blöcke $[\ldots]^{l'}$ existiert eine Grenze, die jedes Element aus $\mathrm{RD}_{entry}(l')$ nach $\mathrm{RD}_{exit}(l')$ lässt.

Nun lassen sich Grenzen aufstellen, die ausdrücken wie der Kontrollfluss verlaufen könnte, für das Beispielprogramm ergibt sich:

$\mathrm{RD}_{entry}(2) \supseteq \mathrm{RD}_{exit}(1)$
$\mathrm{RD}_{entry}(3) \supseteq \mathrm{RD}_{exit}(2)$
$\mathrm{RD}_{entry}(3) \supseteq \mathrm{RD}_{exit}(5)$
$\mathrm{RD}_{entry}(5) \supseteq \mathrm{RD}_{exit}(4)$
$\mathrm{RD}_{entry}(6) \supseteq \mathrm{RD}_{exit}(3)$

Allgemein existiert also eine Grenze $\mathrm{RD}_{entry}(l) \supseteq \mathrm{RD}_{exit}(l')$, wenn der Kontrollfluss von l' auf l übergehen kann. Folglich drückt die Grenze

$$\mathrm{RD}_{entry}(l) \supseteq \{(\mathtt{x}, ?), (\mathtt{y}, ?), (\mathtt{z}, ?)\}$$

aus, dass nicht bekannt ist, wo die nicht initialisierten Variablen definiert werden.
Schnell wird klar, dass die Lösung des Ansatzes mit Gleichungssystem auf Seite 7 auch die Lösung zu obigem Grenzwertsystem ist.
Um diesen Zusammenhang noch deutlicher zu machen, lassen sich alle Grenzen mit der selben linken Seite zusammenfassen, aus

$$\mathrm{RD}_{exit}(1) \supseteq \mathrm{RD}_{entry}(1) \setminus \{(\mathtt{y}, l) \mid l \in \mathbf{Lab}\}$$
$$\mathrm{RD}_{exit}(1) \supseteq \{(\mathtt{y}, 1)\}$$

wird dann zum Beispiel

$$\mathrm{RD}_{exit}(1) \supseteq \mathrm{RD}_{entry}(1) \setminus \{(\mathtt{y}, l) \mid l \in \mathbf{Lab}\} \cup \{(\mathtt{y}, 1)\}$$

und man erhält das gleiche Lösungssystem wie im Ansatz mit Gleichungssystem, nur dass die Gleichheitszeichen durch Mengenoperatoren, genauer „ist (Teil)Menge“, ersetzt wurden.

Zusammenfassend lässt sich also sagen, dass die Ansätze mit Gleichungssystem und Grenzwerten sehr ähnlich sind, auch wenn das nicht immer so leicht ersichtlich ist wie in diesem Beispiel.
Wie gezeigt wurde, setzt dieser Ansatz ein wenig grundlegender an, so dass bei komplexen Programmen das Zusammenfassen aller Schranken mit der selben linken Seite eine stärkere Vereinfachung zur Folge haben könnte als der Ansatz mit Gleichungssystemen.

2.3 Grenzwertbasierte Analyse

Das Ziel der grenzwertbasierten Analyse ist es herauszufinden, von welchem Block auf welchen der Kontrollfluss übergehen könnte. In WHILE ist das natürlich sofort ersichtlich, in komplexeren imperativen, funktionalen oder sogar Objekt-orientierten Programmiersprachen ist das ungleich schwerer festzustellen.
Stellt man sich zum Beispiel folgendes, funktionales Programm vor

```
let  f = fn x => x 1;
     g = fn y => y+2;
     h = fn z => z+3;
in   (f  g)  +  (f  h)
```

ist es schon schwerer zu sehen, wo hier der Kontrollfluss verläuft.
Generell definiert dieses Programm drei Funktionen `f`, `g` und `h`, `f` ist die Hauptfunktion, der `g` und `h` als Parameter `x` übergeben werden. `x` wird in `f` je der Parameter 1 übergeben, so dass das Ergebnis 7 sein wird.
Ein Aufruf von `f` wird die Kontrolle an den Funktionskörper von `f`, also `x 1` übergeben, und dieser Aufruf von `x` wird die Kontrolle wiederum an den Funktionskörper von `x` übergeben, und genau hier wird das Problem klar: Man muss wissen, mit welchem Parameter `f` aufgerufen wurde, um zu wissen auf welches `x` der Kontrollfluss übergeht. Das ist Aufgabe der Kontrollflussanalyse, festzustellen, welche Funktionen durch einen Funktionsaufruf alle aufgerufen werden könnten.
Da sich das Labeln einzelner Blöcke wie in der imperativen WHILE Sprache deutlich schwieriger gestalten würde, da die Blöcke geschachtelt auftreten, wird hier jeder Unterausdruck mit einem eigenen Label versehen

Nimmt man nun folgendes Programm:

$$[[\texttt{fn x => } [\texttt{x}]^1]^2\ [\texttt{fn y => } [\texttt{y}]^3]^4]^5$$

Es ruft die Funktion `fn x => x` mit dem Parameter `fn y => y` auf und lässt `fn y => y` sich selbst aufrufen, hierbei werden alle $[\ldots]^l$ ausgelassen.
Bei dieser Analyse befasst man sich mit den Labels selbst, statt die Ein- und Austrittspunkte zu betrachten, hierbei kann man ausnutzen, dass in dieser einfachen funktionalen Sprache keine Seiteneffekte auftreten. Die Kontrollflussanalyse betrachtet nun Paare $(\hat{C},\hat{p})$ aus Funktionen, wo $\hat{C}(l)$ die Werte enthalten soll, die der Unterausdruck mit dem Label l annehmen könnte und $\hat{p}(x)$ die Werte, die die Variable x annehmen könnte.
Eine Möglichkeit die Kontrollflussanalyse durchzuführen ist es, Grenzwerte zu sammeln, für das obige Beispiel sieht das folgerndermaßen aus, hier werden abstrakten Funktionswerten Labels zugewiesen:

$$\{\text{fn x => [x]}^1\} \subseteq \hat{C}(2)$$

$$\{\text{fn x => [y]}^3\} \subseteq \hat{C}(4)$$

Diese Grenzen stellen dar, dass die Abschätzung einer Funktionsabstraktion einen Grenzwert ergibt, der die abstrahierte Funktion enthält, das zugrundeliegende Muster ist also dass für jeden

Block [fn x => e]^l ein Grenzwert {fn x => e} $\subseteq \hat{C}(l)$ gebildet wird.
Die zweite Sammlung von Grenzwerten verbindet Variablenwerte mit ihren Labels:

$$\hat{p}(\texttt{x}) \subseteq \hat{C}(1)$$

$$\hat{p}(\texttt{y}) \subseteq \hat{C}(3)$$

Diese Grenzen drücken aus, dass jede Variable auf ihren Wert abgeschätzt wird, also existiert für jedes $[x]^l$ eine Grenze $\hat{p}(x) \subseteq \hat{C}(l)$.
Die dritte und letzte Klasse von Grenzen befasst sich mit Funktionsaufrufen: Für jeden Punkt $[e_1\ e_2]^l$ und jede Funktion $[\texttt{fn}\ x\ \texttt{=>}\ e]^{l'}$ die an diesem Punkt aufgerufen werden könnte existieren:
(i) Eine Grenze die zum Ausdruck bringt, dass jeder formale Parameter an dem Punkt mit einem tatsächlichen verbunden wird.
(ii) Eine Grenze, die feststellt, dass jedes Ergebnis das aus einer Analyse des Funktionskörpers hervorgeht, ein mögliches Ergebnis des Funktionsaufrufs ist.

Das Beispielprogramm beinhaltet zwar nur einen Funktionsaufruf $[[\ldots]^2\ [\ldots]^4]^5$, allerdings zwei gibt es hierfür zwei Kandidaten, zum Beispiel ist $\hat{C}(2)$ eine Teilmenge von $\{\texttt{fn x => [x]}^1, \texttt{fn y => [y]}^3\}$. Wenn die Funktion $\texttt{fn x => [x]}^1$ ausgeführt wird, sind die beiden Grenzen $\hat{C}(4) \subseteq \hat{p}(x)$ und $\hat{C}(1) \subseteq \hat{C}(5)$. Diese bedingten Grenzen lassen sich folgendermaßen ausdrücken:

$$\{\text{fn x => [x]}^1\} \subseteq \hat{C}(2) \Rightarrow \hat{C}(4) \subseteq \hat{p}(x)$$

$$\{\text{fn x => [x]}^1\} \subseteq \hat{C}(2) \Rightarrow \hat{C}(1) \subseteq \hat{C}(5)$$

Falls die aufgerufene Funktion allerdings $\texttt{fn y => [y]}^3$ ist, sehen die bedingten Grenzen so aus:

$$\{\text{fn x => [y]}^3\} \subseteq \hat{C}(2) \Rightarrow \hat{C}(4) \subseteq \hat{p}(y)$$

$$\{\text{fn x => [y]}^3\} \subseteq \hat{C}(2) \Rightarrow \hat{C}(3) \subseteq \hat{C}(5)$$

Genau wie bei der Datenflussanalyse des WHILE-Programms ist die Basislösung zu diesem Grenzwert-System von Interesse: Desto kleiner die Wertepaare von $\hat{C}$ und $\hat{p}$ sind, desto genauer kann die Analyse voraussagen, welche Funktionen aufgerufen werden. Folgende Festlegungen von $\hat{C}$ und $\hat{p}$ ergeben eine Lösung für obige Grenzwerte:

$\hat{C}(1) = \{\texttt{fn y => [y]}^3\}$
$\hat{C}(2) = \{\texttt{fn x => [x]}^1\}$
$\hat{C}(3) = \emptyset$
$\hat{C}(4) = \{\texttt{fn y => [y]}^3\}$
$\hat{C}(5) = \{\texttt{fn y => [y]}^3\}$
$\hat{p}(x) = \{\texttt{fn y => [y]}^3\}$
$\hat{p}(y) = \emptyset$

Unter anderem wird ersichtlich, dass die Funktionsabstraktion `fn y => y` keine zugehörige Applikation hat, da $\hat{p}(y) = \emptyset$ und nur die Abstraktion `fn y => y` abgeschätzt werden kann, da $\hat{C}(5) = \{\texttt{fn y => [y]}^3\}$.
Die Grenzwerteanalyse ist (wie der Name schon sagt) dem Ansatz mit Grenzwerten der Datenflussanalyse sehr ähnlich, in beiden Fällen wird die syntaktische Struktur eines Programmes durch Grenzwerte ausgedrückt und die Basislösung zu diesen Systemen gesucht. Der größte Unterschied zwischen beiden Analysen ist die deutlich komplexere Struktur dieser Grenzwertsysteme in der Grenzwertanalyse, die durch die Funktionsweise funktionaler Programmiersprachen notwendig wird.

2.4 Abstrakte Interpretation

Hier wird, so [2], ein Programm auf der Ebene abstrakter Werte interpretiert, zum Beispiel „gerade“ und „ungerade“ oder „negativ“, „null“ und „postitiv“ statt Integer-Werten.
Diese Analyse wird vor allem zur Verifikation von Programmen eingesetzt.
Die Theorie der Abstrakten Interpretation beruht darauf, die Analyse tatsächlich auszurechnen statt sie nur festzulegen, und sich auf einen folgenden Beweis der Rechnung zu verlassen.

Zu Beginn wird eine Sammelsemantik (collecting semantics) formuliert, die eine Menge von Traces *tr* beinhaltet, die einem Programmpunkt zugeordmet werden:

$$tr \in \textbf{Trace} = (\textbf{Var} \times \textbf{Lab})^*$$

Die Traces werden den Punkten im Programm zugeordnet, an denen während des Ausführung Variablen Werte zugewiesen werden, für das Programm zur Fakultätsberechnung

$$[\mathtt{y} := \mathtt{x}]^1;\ [\mathtt{z} := 1]^2;\ \mathtt{while}\ [\mathtt{y} > 1]^3\ \mathtt{do}\ ([\mathtt{z} := \mathtt{z} * \mathtt{y}]^4;\ [\mathtt{y} := \mathtt{y} - 1]^5);\ [\mathtt{y} := 0]^6$$

ergibt sich beispielweise folgende Trace:

$$((\mathtt{x}, ?), (\mathtt{y}, ?), (\mathtt{z}, ?), (\mathtt{y}, 1), (\mathtt{z}, 2), (\mathtt{z}, 4), (\mathtt{y}, 5), (\mathtt{z}, 4), (\mathtt{y}, 5), (\mathtt{y}, 6))$$

wenn zur Laufzeit der Inhalt der `while`-Schleife zweimal ausgeführt wird.
Diese Traces enthalten genug Informationen, um eine Menge von semantischen Zuweisungen darzustellen:

$$\mathrm{SRD}(tr)(x) = l \iff \text{für das am weitesten rechts gelegene Paar } (x, l') \text{ in } tr \text{ gilt } l = l'$$

Es wird nun DOM(*tr*) für die Menge aller Variablen, für die SRD(*tr*) definiert ist, also:

$$x \in \mathrm{DOM}(tr) \iff \text{es gibt ein Paar } (x, l) \text{ das in } tr \text{ vorkommt}$$

Um zu gewährleisten dass die Zuweisungsanalyse korrekte Ergebnisse liefert, muss sie die semantischen Zuweisungen festhalten: Wenn *tr* eine mögliche Trace kurz vor dem Eintritt in den Block mit dem Label *l* ist, muss gelten:

$$\forall x \in \mathrm{DOM}(tr) : (x, \mathrm{SRD}(tr)(x)) \in \mathrm{RD}_{entry}(l)$$

So ist sicher, dass die Informationen in $\mathrm{RD}_{entry}(l)$, welche Definitionen den Eingangspunkt zu *l* erreichen könnten, richtig sind.
Die Sammelsemantik definiert eine Menge von möglichen Taces für verschiedene Punkte im Programm, das lässt sich wiederrum als Zwölftupel von Elementen aus $(P(\textbf{Trace}))^{12}$ in einem Gleichungssystem darstellen. Die ersten Gleichungen

$$\mathrm{CS}_{exit}(1) = \{tr : (\mathtt{y}, 1) \mid tr \in \mathrm{CS}_{entry}(1)\}$$

$$\begin{aligned}
&\mathrm{CS}_{exit}(2) = \{tr \,:\, (\mathtt{z},2) \mid tr \in \mathrm{CS}_{entry}(2)\}\\
&\mathrm{CS}_{exit}(3) = \mathrm{CS}_{entry}(3)\\
&\mathrm{CS}_{exit}(4) = \{tr \,:\, (\mathtt{z},4) \mid tr \in \mathrm{CS}_{entry}(4)\}\\
&\mathrm{CS}_{exit}(5) = \{tr \,:\, (\mathtt{y},5) \mid tr \in \mathrm{CS}_{entry}(5)\}\\
&\mathrm{CS}_{exit}(6) = \{tr \,:\, (\mathtt{y},6) \mid tr \in \mathrm{CS}_{entry}(6)\}
\end{aligned}$$

stellen dar, wie eine Zuweisung die Erweiterung der Traces hervorruft. $tr : (x,l)$ steht für die Aufnahme des Elements (x,l) zur Traceliste tr, also gilt dass $((x_1,l_1), \ldots ,(x_n,l_n)) : (x,l)$ gleichbedeutend mit $((x_1,l_1), \ldots ,(x_n,l_n),(x,l))$ ist.
Außerdem gibt es folgende Gleichungen

$$\begin{aligned}
\mathrm{CS}_{entry}(2) &= \mathrm{CS}_{exit}(1)\\
\mathrm{CS}_{entry}(3) &= \mathrm{CS}_{exit}(2) \cup \mathrm{CS}_{exit}(5)\\
\mathrm{CS}_{entry}(4) &= \mathrm{CS}_{exit}(3)\\
\mathrm{CS}_{entry}(5) &= \mathrm{CS}_{exit}(4)\\
\mathrm{CS}_{entry}(6) &= \mathrm{CS}_{exit}(3)
\end{aligned}$$

die den Kontrollfluss im Programm darstellen. Mit genaueren Informationen über die Werte der Variablen ließen sich die Mengen $\mathrm{CS}_{entry}(4)$ und $\mathrm{CS}_{entry}(6)$ genauer darstellen, aber obige Definitionen sind ausreichend, um den Ansatz darzustellen.
Schließlich nimmt man noch

$$\mathrm{CS}_{entry}(2) = \{((\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?))\}$$

um zum Ausdruck zu bringen, dass am Anfang alle Variablen nicht initialisiert sind.
Wie in den Kapiteln zuvor lässt sich das obige Gleichungssystem in folgender Form darstellen:

$$\overrightarrow{\mathrm{CS}} = G(\overrightarrow{\mathrm{CS}})$$

wobei $\overrightarrow{\mathrm{CS}}$ ein Zwölftupel von Elementen aus $(P(\mathbf{Trace}))^{12}$ und G eine monotone Funktion folgender Art ist:

$$G : (P(\mathbf{Trace}))^{12} \rightarrow P(\mathbf{Trace}))^{12}$$

Durch mathematische Gesetzmäßigkeiten (die im Anhang A im Buch [1] erklärt werden) ist sichergestellt, dass eine Basislösung $lfp(G)$ existiert, da $(P(\mathbf{Trace}))^{12}$ aber nicht endlich ist lassen sich die vorherigen Methoden diese herauszufinden nicht ohne weiteres anwenden.

Da die Sammelsemantik mit Traces, die Zuweisungsanalyse aber mit Paaren aus Variablen und Labels arbeitet, muss ein Weg gefunden werden, diese in Relation zu setzen. Hierzu verwendet man eine Funktionsabstraktion α und eine Konkretisierung γ, die in folgendem Zusammenhang stehen:

$$P(\mathbf{Trace}) \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} P(\mathbf{Var} \times \mathbf{Lab})$$

Hierbei ist die Idee, dass die Funktionsabstraktion α die Informationen über die Reichweite einer Zuweisung aus einer Menge von Traces extrahiert:

$$\alpha(X) = \{(x, \text{SRD}(tr)(x)) \mid x \in \text{DOM}(tr) \wedge tr \in X\}$$

Die Funktionskonkretisierung γ erstellt dann alle Traces tr, die zu den Reichweiten der Zuweisungen passen:

$$\gamma(Y) = \{tr \mid \forall x \in \text{DOM}(tr) : (x, \text{SRD}(tr)(x)) \in Y\}$$

Meist muss für α und γ gelten:

$$\alpha(X) \subseteq Y \iff X \subseteq \gamma(Y)$$

so dass man sagen kann dass (α, γ) eine Adjunktion bzw. Galoisverbindung ist.

Nun soll gezeigt werden, wie die Sammelsemantik genutzt werden kann, um die Analyse zu berechnen, diese Analyse wird *Induzierte Analyse* genannt. Zuerst müssen

$$\overrightarrow{\alpha}(X_1, \ldots, X_{12}) = (\alpha(X_1), \ldots, \alpha(X_{12}))$$

$$\overrightarrow{\gamma}(Y_1, \ldots, Y_{12}) = (\gamma(Y_1), \ldots, \gamma(Y_{12}))$$

definiert werden, wobei eine Funktion $\overrightarrow{\alpha} \circ G \circ \overrightarrow{\gamma}$ verwendet wird, die folgendermaßen wirkt:

$$(\overrightarrow{\alpha} \circ G \circ \overrightarrow{\gamma}) : (P(\mathbf{Var} \times \mathbf{Lab}))^{12} \to (P(\mathbf{Var} \times \mathbf{Lab}))^{12}$$

Diese Funktion definiert indirekt eine Zuweisungsanalyse; da G durch eine Menge von Gleichungen über $P(\mathbf{Trace})$ spezifiziert ist, kann man $\overrightarrow{\alpha} \circ G \circ \overrightarrow{\gamma}$ verwenden, um eine neue Menge von Gleichungen über $P(\mathbf{Var} \times \mathbf{Lab})$ zu berechnen. Hier das Beispiel für die Gleichung

$$\text{CS}_{exit}(4) = \{tr : (\mathsf{z}, 4) \mid tr \in \text{CS}_{entry}(4)\}$$

der zugehörige Teil in der Definition ist:

$$G_{exit}(4)(\ldots, \text{CS}_{entry}(4), \ldots) = \{tr : (\mathsf{z}, 4) \mid tr \in \text{CS}_{entry}(4)\}$$

So dass sich der zugehörige Teil in der Definition von $\overrightarrow{\alpha} \circ G \circ \overrightarrow{\gamma}$ berechnen lässt:

$$\begin{aligned}
&\alpha(G_{exit}(4)\overrightarrow{\gamma}(\ldots, \text{RD}_{entry}(4), \ldots))) \\
&= \alpha(\{tr : (\mathsf{z}, 4) \mid tr \in \gamma(\text{RD}_{entry}(4))\}) \\
&= \{(x, \text{SRD}(tr : (\mathsf{z}, 4))(x) \mid x \in \text{DOM}(tr : (\mathsf{z}, 4))), \forall y \in \text{DOM}(tr) : (y, \text{SRD}(tr)(y)) \in \text{RD}_{entry}(4)\} \\
&= \{(x, \text{SRD}(tr : (\mathsf{z}, 4))(x) \mid x \neq \mathsf{z}, x \in \text{DOM}(tr : (\mathsf{z}, 4))), \forall y \in \text{DOM}(tr) : (y, \text{SRD}(tr)(y)) \in \text{RD}_{entry}(4)\} \\
&\quad \cup \{(x, \text{SRD}(tr : (\mathsf{z}, 4))(x) \mid x = \mathsf{z}, x \in \text{DOM}(tr : (\mathsf{z}, 4))), \forall y \in \text{DOM}(tr) : (y, \text{SRD}(tr)(y)) \in \text{RD}_{entry}(4)\} \\
&= \{(x, \text{SRD}(tr)(x)) \mid x \neq \mathsf{z}, x \in \text{DOM}(tr), \forall y \in \text{DOM}(tr) : (y, \text{SRD}(tr)(y)) \in \text{RD}_{entry}(4)\} \\
&\quad \cup \{(\mathsf{z}, 4) \, \forall y \in \text{DOM}(tr) : (y, \text{SRD}(tr)(y)) \in \text{RD}_{entry}(4)\} \\
&= (\text{RD}_{entry}(4) \backslash \{(\mathsf{z}, l) \mid l \in \mathbf{Lab}\}) \cup \{(\mathsf{z}, 4)\}
\end{aligned}$$

Die resultierende Gleichung

$$\mathrm{RD}_{exit}(4) = (\mathrm{RD}_{entry}(4) \backslash \{(\mathsf{z}, l) \mid l \in \mathbf{Lab}\}) \cup \{(\mathsf{z}, 4)\}$$

ist das gleiche Ergebnis wie das der Datenflussanalyse auf Seite 10, die selben Rechenschritte können auch auf die anderen Gleichungen angewendet werden.

Wie bereits erwähnt, existiert eine Basislösung für die Gleichung

$$\overrightarrow{\mathrm{RD}} = (\overrightarrow{\alpha} \circ G \circ \overrightarrow{\gamma})(\overrightarrow{\mathrm{RD}})$$

die $lfp(\overrightarrow{\alpha} \circ G \circ \overrightarrow{\gamma})$ heißt, hierzu müssen **Var** und **Lab** durch die endlichen Mengen $\mathbf{Var}_*$ und $\mathbf{Lab}_*$ ersetzt werden, dann ist die Lösung $(\overrightarrow{\alpha} \circ G \circ \overrightarrow{\gamma})^n(\overrightarrow{\emptyset})$.

2.5 Typ- und Effektsysteme

Hier werden, so [2], Programmen *Typen* zugewiesen, um Laufzeitfehler auszuschließen.
Dieses Verfahren lässt sich besonders gut auf funktionale Programmiersprachen anwenden.

Um sich Typ- und Effektsysteme besser vorstellen zu können, sollte von einer typisierten imperativen oder funktional Programmiersprache ausgegangen werden, hierzu wird die While-Sprache um folgendes erweitert: Ein Statement S stellt den Übergang von einem Zustand in einen anderen dar (wenn das Statement terminiert), von daher kann davon ausgegangen werden, dass es den Typ $\Sigma \to \Sigma$ hat, wobei Σ der Typ der Zustände ist, das kann folgendermaßen ausgedrückt werden:

$$S : \Sigma \to \Sigma$$

Um dies zu formalisieren sollen folgende Axiome gelten:

$$[x := a]^l : \Sigma \to \Sigma$$

$$[\texttt{skip}] : \Sigma \to \Sigma$$

$$\frac{S_1 : \Sigma \to \Sigma \quad S_2 : \Sigma \to \Sigma}{S_1;S_2 : \Sigma \to \Sigma}$$

$$\frac{S_1 : \Sigma \to \Sigma \quad S_2 : \Sigma \to \Sigma}{\texttt{if}\ [b]^l\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \Sigma \to \Sigma}$$

$$\frac{S : \Sigma \to \Sigma}{\texttt{while}\ [b]^l\ \texttt{do}\ S : \Sigma \to \Sigma}$$

Meist kann ein Typ- und Effektystem als die Vereinigung zweier Komponenten gesehen werden, einem Effektsystem und einem Annotierten Typsystem. Im Effektsystem kommen typischerweise Ausdrücke der Gestalt $S : \Sigma \xrightarrow{\varphi} \Sigma$ vor, wobei der Effekt φ ausdrückt, was bei der Ausführung von S passiert, das könnten Fehler, Exceptions oder Dateizugriffe sein. Das Annotierte Typsystem besteht aus Ausdrücken der Form $S : \Sigma_1 \to \Sigma_2$, wobei Σ_i die Eigenschaften von Zuständen (zum Beispiel ob eine Variable einen positiven Wert hat) beschreibt.

2.5.1 Annotierte Typsysteme

Annotierte Typsysteme lassen sich am Beispiel der WHILE-Sprache demonstrieren.
Mit den obigen Axiomen und der Annahme, dass bei Ausdrücken der Gestalt $S : \mathrm{RD}_1 \to \mathrm{RD}_2$ $\mathrm{RD}_1, \mathrm{RD}_2 \in P(\mathbf{Var} \times \mathbf{Lab})$ gilt, lassen sich folgende detailierteren Axiome aufstellen:

$[ass]$ $\quad [x := a]^{l'} : \mathrm{RD} \to ((\mathrm{RD} \backslash \{x, l) \mid l \in \mathbf{Lab}\}) \cup \{(x, l')\})$

$[skip]$ $\quad [\mathtt{skip}]^{l'} : \mathrm{RD} \to \mathrm{RD}$

$$[seq] \quad \frac{S_1 : \mathrm{RD}_1 \to \mathrm{RD}_2 \quad S_2 : \mathrm{RD}_2 \to \mathrm{RD}_3}{S_1; S_2 : \mathrm{RD}_1 \to \mathrm{RD}_3}$$

$$[if] \quad \frac{S_1 : \mathrm{RD}_1 \to \mathrm{RD}_2 \quad S_2 : \mathrm{RD}_2 \to \mathrm{RD}_3}{\mathtt{if}[b]^l \mathtt{then} S_1 \mathtt{else} S_2 : \mathrm{RD}_1 \to \mathrm{RD}_2}$$

$$[wh] \quad \frac{S : \mathrm{RD} \to \mathrm{RD}}{\mathtt{while}[b]^l \mathtt{do} S : \mathrm{RD} \to \mathrm{RD}}$$

$$[sub] \quad \frac{S : \mathrm{RD}_2 \to \mathrm{RD}_3}{S : \mathrm{RD}_1 \to \mathrm{RD}_4} \; wenn \; \mathrm{RD}_1 \subseteq \mathrm{RD}_2 \; und \; \mathrm{RD}_3 \subseteq \mathrm{RD}_4$$

Also lässt sich $S : \mathrm{RD}_1 \to \mathrm{RD}_2$ folgendermaßen ausdrücken:

$$\mathrm{RD}_1 \subseteq \mathrm{RD}_{entry}(init(S))$$

$$\forall l \in final(S) : \mathrm{RD}_{exit}(l) \subseteq \mathrm{RD}_2$$

Wendet man diese Regeln nun auf das Programm zur Fakultätsberechnung

$$[\mathtt{y} := \mathtt{x}]^1; [\mathtt{z} := 1]^2; \mathtt{while}\ [\mathtt{y} > 1]^3\ \mathtt{do}\ ([\mathtt{z} := \mathtt{z} * \mathtt{y}]^4; [\mathtt{y} := \mathtt{y} - 1]^5); [\mathtt{y} := 0]^6$$

an, und schreibt RD_f für $\{(\mathtt{x}, ?), (\mathtt{y}, 1), (\mathtt{y}, 5), (\mathtt{z}, 2), (\mathtt{z}, 4)\}$, ergibt sich nach $[ass]$

$[\mathtt{z} := \mathtt{z} * \mathtt{y}]^4 : \mathrm{RD}_f \to \{(\mathtt{x}, ?), (\mathtt{y}, 1), (\mathtt{y}, 5), (\mathtt{z}, 4)\}$
$[\mathtt{y} := \mathtt{y} - 1]^5 : \{(\mathtt{x}, ?), (\mathtt{y}, 1), (\mathtt{y}, 5), (\mathtt{z}, 4)\} \to \{(\mathtt{x}, ?), (\mathtt{y}, 5), (\mathtt{z}, 4)\}$

so dass $[seq]$

$$([\mathtt{z} := \mathtt{z} * \mathtt{y}]^4; [\mathtt{y} := \mathtt{y} - 1]^5) : \mathrm{RD}_f \to \{(\mathtt{x}, ?), (\mathtt{y}, 5), (\mathtt{z}, 4)\}$$

ergibt. Da nun $\{(\mathtt{x}, ?), (\mathtt{y}, 5), (\mathtt{z}, 4)\} \in \mathrm{RD}_f$, ergibt sich

$$([\mathtt{z} := \mathtt{z} * \mathtt{y}]^4; [\mathtt{y} := \mathtt{y} - 1]^5) : \mathrm{RD}_f \to \mathrm{RD}_f$$

auf das sich $[wh]$ anwenden lässt:

$$\mathtt{while}\ [\mathtt{y} > 1]^3\ \mathtt{do}\ ([\mathtt{z} := \mathtt{z} * \mathtt{y}]^4; [\mathtt{y} := \mathtt{y} - 1]^5) : \mathrm{RD}_f \to \mathrm{RD}_f$$

mit $[ass]$ erhält man nun

$$[\mathtt{y} := \mathtt{x}]^1 \;:\; \{(\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?)\} \rightarrow \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},?)\}$$
$$[\mathtt{z} := 1]^2 \;:\; \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},?)\} \rightarrow \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},2)\}$$
$$[\mathtt{y} := 0]^6 \;:\; \mathrm{RD}_f \rightarrow \{(\mathtt{x},?),(\mathtt{y},6),(\mathtt{z},2),(\mathtt{z},4)\}$$

Da $\{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},2)\} \subseteq \mathrm{RD}_f$, lassen sich [*seq*] und [*sub*] anwenden:

$$([\mathtt{y} := \mathtt{x}]^1; [\mathtt{z} := 1]^2; \mathtt{while}\ [\mathtt{y} > 1]^3\ \mathtt{do}\ ([\mathtt{z} := \mathtt{z} * \mathtt{y}]^4; [\mathtt{y} := \mathtt{y} - 1]^5); [\mathtt{y} := 0]^6) :$$

$$\{(\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?)\} \rightarrow \{(\mathtt{x},?),(\mathtt{y},6),(\mathtt{z},2),(\mathtt{z},4)\}$$

was wiederum dem Ergebnis in der Tabelle der Zuweisungsanalyse auf Seite 5 entspricht.

2.5.2 Effektsysteme

Das Effektsystem lässt sich gut an der bereits aufgestellten funktionalen Sprache veranschaulichen; die Vorgehensweise besteht hier darin, ein Typsystem wie im Abschnitt vorher mit Informationen aus der Analyse zu verbinden.
Für ein einfaches Beispiel werden Variablen x, Funktionsabstraktionen $\texttt{fn}_\pi \texttt{ => } e$ und Funktionsapplikationen $e_1\ e_2$ benötigt, Ausdrücke folgen dann der Form

$$\Gamma \vdash e : \tau$$

wo Γ eine Typumgebung ist, die jeder Variable freien Variable von e Typen zuweist, und τ der Typ von e ist. Zur Vereinfachung wird hier davon ausgegangen, dass nur Basistypen wie `int` oder `bool` sowie Funktionstypen der Form $\tau_1 \rightarrow \tau_2$ existieren. Das Typsystem ist durch folgende Axiome festgelegt:

$$\Gamma \vdash x : \tau_x \ \textit{wenn}\ \Gamma(x) = \tau_x$$

$$\frac{\Gamma[x \mapsto \tau_x] \vdash e : \tau}{\Gamma \vdash \texttt{fn}_\tau\ x \texttt{ => } e : \tau_x \rightarrow \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau, \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$$

Das Axiom der Variablen drückt aus, dass der Typ von x durch eine Schätzung der Typumgebung festgelegt wird.
Die Regel der Funktionsabstraktion verlangt eine Abschätzung eines Typs τ_x für x, so lässt sich gleichzeitig ein Typ für den Körper der Funktionsabstraktion abschätzen.
Das dritte Axiom, das sich auf die Funktionsapplikation bezieht, erwartet dass die Typen für den Operator und den Parameter herausgefunden werden, und legt implizit fest dass der Operator einen Funktionstyp hat, da $e_1 : \tau_2 \rightarrow \tau$.
τ_2 bringt zum Ausdruck, dass der Typ des Parameters dem für diesen Funktionsparameter erwarteten entsprechen muss.

Zur Veranschaulichung wird noch einmal das Beispielprogramm der funktionalen Sprache

$$(\texttt{fn}_\texttt{x}\ \texttt{x => x})(\texttt{fn}_\texttt{y}\ \texttt{y => y})$$

verwendet, wobei die Funktion `fn x => x` den Namen X und die Funktion `fn y => y` den Namen Y erhalten. Um festzustellen, dass die Funktion den Typ `int` $\rightarrow$ `int` besitzt, wird zuerst festgestellt, dass $[\texttt{y} \mapsto \texttt{int}] \vdash \texttt{y : int}$, so dass:

$$[\,] \vdash \texttt{fn}_\texttt{y}\ \texttt{y => y : int} \rightarrow \texttt{int}$$

Außerdem gilt $[\texttt{x} \mapsto \texttt{int} \mapsto \texttt{int}] \vdash \texttt{x : int} \rightarrow \texttt{int}$, also:

$$[\,] \vdash \texttt{fn}_{\texttt{x}}\ \texttt{x => x : (int} \rightarrow \texttt{int)} \rightarrow \texttt{(int} \rightarrow \texttt{int)}$$

Aus der Regel der Funktionsapplikation ergibt sich dann:

$$[\,] \vdash (\texttt{fn}_{\texttt{x}}\ \texttt{x => x})(\texttt{fn}_{\texttt{y}}\ \texttt{y => y}) : \texttt{int} \rightarrow \texttt{int}$$

Eine Möglichkeit der Analyse ist eine sogenannte Call-Tracking Analysis, also eine Analyse die Aufrufen nachgeht. Hierzu wird für jeden Unterausdruck herausgefunden, welche Funktionsabstraktionen während der Auswertung aufgerufen werden könnten.
Um die Effekte der Unterausdrücke festzustellen, werden die Funktionstypen mit ihrem Effekt annotiert, so bedeutet $\texttt{int} \xrightarrow{\{X\}} \texttt{int}$ dass die entsprechende Funktion Integer auf Integer abbildet und über den Effekt $\{X\}$ verfügt, also dass während der Funktionsausführung die Funktion X aufgerufen werden könnte. Also sind die annotierten Typen $\hat{\tau}$ entweder Basistypen oder haben die Form

$$\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2$$

wo φ der Effekt, also der Name der Funktionsabstraktion die aufgerufen werden könnte, ist.

2.6 Algorithmen

In der Datenfluss- und Grenzwertbasierten Analyse ist es notwendig, die Basislösung zu berechnen. Hierzu wurden die Zwölftupel $\overrightarrow{\mathrm{RD}} \in (P(\mathbf{Var}_* \times \mathbf{Lab}_*))^{12}$ von Mengen von Paaren aus Variablen und Labels verwendet, wo das Label sich auf einen Block bezieht, in dem der Variable zuletzt ein Wert zugewiesen wurde. Durch ein Gleichungs- oder Grenzwertsystem mit der Gleichung $\overrightarrow{\mathrm{RD}} = F(\overrightarrow{\mathrm{RD}})$ beziehungsweise der Grenze $\overrightarrow{\mathrm{RD}} \sqsupseteq F(\overrightarrow{\mathrm{RD}})$, wo F eine monotone Funktion über $(P(\mathbf{Var}_* \times \mathbf{Lab}_*))^{12}$ ist, lässt sich die Lösung $F^n(\overrightarrow{\emptyset})$ für ein beliebiges n berechnen.
Eine Auswertung dieser Systeme ohne einen geeigneten Algorithmus artet sehr schnell in sehr viel Arbeit aus, darum soll hier die *Chaotische Iteration* vorgestellt werden, der Herzstück vieler hierfür geeigneter Algorithmen ist:
Hierzu schreibt man

$$\begin{aligned} \overrightarrow{\mathrm{RD}} &= (\mathrm{RD}_1, \ldots, \mathrm{RD}_{12}) \\ F(\overrightarrow{\mathrm{RD}}) &= (F_1(\overrightarrow{\mathrm{RD}}), \ldots, F_{12}(\overrightarrow{\mathrm{RD}})) \end{aligned}$$

und verwendet folgenden Algorithmus:

INPUT:	Beispielgleichungen der Reichweitenanalyse	
OUTPUT:	Die Basislösung $\overrightarrow{\mathrm{RD}} = (\mathrm{RD}_1, \ldots, \mathrm{RD}_{12})$	
METHOD:	Schritt 1:	Initialisierung
		$\mathrm{RD}_1 := \emptyset; \ldots; \mathrm{RD}_{12} := \emptyset$
	Schritt 2:	Iteration
		`while` $\mathrm{RD}_j \neq F_j(\mathrm{RD}_1, \ldots, \mathrm{RD}_{12})$ `for some` j
		`do` $\mathrm{RD}_j := F_j(\mathrm{RD}_1, \ldots, \mathrm{RD}_{12})$

Dieser Algorithmus terminiert und gibt einen Punkt von F zurück, der eine Lösung zu der übergebenen Gleichung ist.
Da $\overrightarrow{\emptyset} \sqsubseteq \overrightarrow{\mathrm{RD}} \sqsubseteq F(\overrightarrow{\mathrm{RD}}) \sqsubseteq F^n(\overrightarrow{\emptyset})$ alle Punkte des Gleichungssystems beinhaltet, erhält man durch den Algorithmus nicht nur einen beliebigen Fixpunkt von F, sondern genau den kleinsten Fixpunkt, also die Basislösung.
Um kurz zu zeigen, dass der Algorithmus tatsächlich terminiert, geht man davon aus dass wenn $j\ \mathrm{RD}_j \neq F_j(\mathrm{RD}_1, \ldots, \mathrm{RD}_{12})$ erfüllt, $\mathrm{RD}_j \subset F_j(\mathrm{RD}_1, \ldots, \mathrm{RD}_{12})$ gilt, so wächst $\overrightarrow{\mathrm{RD}}$ bei jeder Iteration mindestens um eins. Da festgelegt wurde, dass $(P(\mathbf{Var}_* \times \mathbf{Lab}_*))^{12}$ endlich ist, terminiert folglich der Algorithmus.

2.7 Transformationen

Ein wichtiges Aufgabengebiet der Programmanalyse ist es, das Programm auf Quellcodeebene oder während einem Zwischenschritt der Kompilierung zu transformieren, um es performanter zu machen. Um dies zu veranschaulichen, kann man die Zuweisungsanalyse verwenden, um eine Transformation namens *Constant Folding* durchzuführen. Dazu gehören zwei Vorgehensweisen: Erstens werden alle Variablen, die sich im Verlauf der Programmausführung nie ändern werden, durch Konstanten ersetzt, zweitens werden Ausdrücke vereinfacht, indem sie teilweise ausgewertet werden, alle Unterausdrücke, die auf Variablen zugreifen, können schon vor der eigentlichen Ausführung ausgewertet werden.
Nimmt man ein Programm S^*, zu dem RD eine Lösung, idealerweise die Basislösung, der Reichweitenanalyse ist. Für jeden Unterausdruck S von S^* soll nun beschrieben werden, wie er in ein besseres (performanteres) Statement S' transformiert werden kann, für einen Schritt schreibt man

$$\mathrm{RD} \vdash S \rhd S'$$

Für die Transformationen kann auf folgende Regeln und Axiome zurückgegriffen werden:

$[ass_1]$ $\quad \mathrm{RD} \vdash [x := a]^l \rhd [x := a[y \mapsto n]]^l$

$$\text{if} \begin{cases} y \in FV(a) \wedge (y, ?) \notin \mathrm{RD}_{entry}(l) \wedge \\ \forall (z, l') \in \mathrm{RD}_{entry}(l) : (z = y \Rightarrow [\ldots]^{l'} \text{ is } [y := n]^{l'} \end{cases}$$

$[ass_2]$ $\quad \mathrm{RD} \vdash [x := a]^l \rhd [x := n]^l$
if $FV(a) = \emptyset \wedge a \notin \mathbf{Num} \wedge a$ evaluates to n

$$[seq_1] \quad \frac{\mathrm{RD} \vdash S_1 \rhd S_1'}{\mathrm{RD} \vdash S_1; S_2 \rhd S_1'; S_2}$$

$$[seq_2] \quad \frac{\mathrm{RD} \vdash S_2 \rhd S_2'}{\mathrm{RD} \vdash S_1; S_2 \rhd S_1; S_2'}$$

$$[if_1] \quad \frac{\mathrm{RD} \vdash S_1 \rhd S_1'}{\mathrm{RD} \vdash \texttt{if}\ [b]^l\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \rhd \texttt{if}\ [b]^l\ \texttt{then}\ S_1'\ \texttt{else}\ S_2}$$

$$[if_2] \quad \frac{\mathrm{RD} \vdash S_2 \rhd S_2'}{\mathrm{RD} \vdash \texttt{if}\ [b]^l\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \rhd \texttt{if}\ [b]^l\ \texttt{then}\ S_1\ \texttt{else}\ S_2'}$$

$$[wh] \quad \frac{\mathrm{RD} \vdash S \rhd S'}{\mathrm{RD} \vdash \texttt{while}\ [b]^l\ \texttt{do}\ S \rhd \texttt{while}\ [b]^l\ \texttt{do}\ S'}$$

Das erste Axiom, $[ass_1]$, drückt den ersten Schritt aus: Variablen, die sich nicht verändern, können durch Konstanten ersetzt werden. $[ass_2]$ beschreibt den zweiten Schritt, Teilausdrücke können ausgewertet werden, wenn keine Variablen darin vorkommen, da der Teilausdruck dann immer das gleiche ergeben wird.

Die letzten fünf Regeln beschreiben die Transformation, zum Beispiel dass wenn ein Unterausdruck transformiert werden kann der ganze Ausdruck transformiert werden kann. Allerdings ist keine feste Reihenfolge der Transformationen vorgegeben, so dass es dafür viele unterschiedliche Möglichkeiten gibt. RD $\vdash \cdot \rhd \cdot$ ist weder transitiv noch reflexiv, ganz einfach weil das nirgends festgelegt wurde, also können ohne weitere Bedenken viele Transformationen in beliebiger Reihenfolge erfolgen.
Um eine Transformation zu veranschaulichen soll folgendes simple Programm dienen:

$$[\mathtt{x} := 10]^1; [\mathtt{y} := \mathtt{x} + 10]^2; [\mathtt{z} := \mathtt{y} + 10]^3$$

Eine Reichweitenanalyse ergibt folgendes:

$$\begin{aligned}
\mathrm{RD}_{entry}(1) &= \{(\mathtt{x}, ?), (\mathtt{y}, ?), (\mathtt{z}, ?)\} \\
\mathrm{RD}_{exit}\ (1) &= \{(\mathtt{x}, 1), (\mathtt{y}, ?), (\mathtt{z}, ?)\} \\
\mathrm{RD}_{entry}(2) &= \{(\mathtt{x}, 1), (\mathtt{y}, ?), (\mathtt{z}, ?)\} \\
\mathrm{RD}_{exit}\ (2) &= \{(\mathtt{x}, 1), (\mathtt{y}, 2), (\mathtt{z}, ?)\} \\
\mathrm{RD}_{entry}(3) &= \{(\mathtt{x}, 1), (\mathtt{y}, 2), (\mathtt{z}, ?)\} \\
\mathrm{RD}_{exit}\ (3) &= \{(\mathtt{x}, 1), (\mathtt{y}, 2), (\mathtt{z}, 3)\}
\end{aligned}$$

Aus $[ass_1]$ ergibt sich

$$\mathrm{RD} \vdash [\mathtt{y} := \mathtt{x} + 10]^2 \rhd [\mathtt{y} := 10 + 10]^2$$

folglich ergibt sich aus den weiteren Regeln

$$\mathrm{RD} \vdash [\mathtt{x} := 10]^1; [\mathtt{y} := \mathtt{x} + 10]^2; [\mathtt{z} := \mathtt{y} + 10]^3 \rhd [\mathtt{x} := 10]^1; [\mathtt{y} := 10 + 10]^2; [\mathtt{z} := \mathtt{y} + 10]^3$$

Wenn man nun einfach weitermacht ergibt sich folgende Transformationssequenz:

$$\begin{aligned}
\mathrm{RD} \vdash\ & [\mathtt{x} := 10]^1;\ [\mathtt{y} := \mathtt{x} + 10]^2;\ \ [\mathtt{z} := \mathtt{y} + 10]^3 \\
\rhd\ & [\mathtt{x} := 10]^1;\ [\mathtt{y} := 10 + 10]^2;\ [\mathtt{z} := \mathtt{y} + 10]^3 \\
\rhd\ & [\mathtt{x} := 10]^1;\ [\mathtt{y} := 20]^2;\ \ \ \ \ [\mathtt{z} := \mathtt{y} + 10]^3 \\
\rhd\ & [\mathtt{x} := 10]^1;\ [\mathtt{y} := 20]^2;\ \ \ \ \ [\mathtt{z} := 20 + 10]^3 \\
\rhd\ & [\mathtt{x} := 10]^1;\ [\mathtt{y} := 20]^2;\ \ \ \ \ [\mathtt{z} := 30]^3
\end{aligned}$$

3 Zusammenfassung

In dieser Arbeit wurde ein grober Überblick über einige der Ansätze zu einer Programmanalyse gegeben. Obwohl diese Ansätze an sich relativ verschieden sind lag der Fokus darauf, die Gemeinsamkeiten herauszuarbeiten: Jeder der vorgestellten Ansätze lässt sich auf einen der zwei Typen, Gleichungssystem-basiert und Grenzwertsystem-basiert zurückführen.
Zusammenfassend lässt sich sagen, dass Programmanalysen ein wichtiger Punkt in der Erstellung von ausführbarem Binärcode sind, und dass die Komplexität der Analysevorgänge mit der Komplexität der verwendeten Programmiersprache in direktem Zusammenhang stehen.

Literatur

[1] F. Nielson, H. R. Nielson, C. Hankin: Principles of Program Analysis. Springer (Corrected 2nd printing, 452 pages, ISBN 3-540-65410-0), 2005.

[2] B. König: Vorlesung „Programmanalyse“ Sommersemester 2008. Universität Duisburg-Essen, Fachgebiet für Theoretische Informatik, 2008.