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

Betreuer:	Dr. Martin Lange
Verantw. Hochschullehrer:	Prof. Martin Hofmann, Ph.D.

Gliederung

1. Einleitung
2. Hauptteil
   - Verwendete Syntax
     - Zuweisungsanalyse
   - Datenflussanalyse
     - Ansatz mit Gleichungssystem
     - Ansatz mit Grenzwerten
   - Grenzwertbasierte Analyse
   - Abstrakte Interpretation
   - Typ- und Effektsysteme
     - Annotierte Typsysteme
     - Effektsysteme
   - Transformationen
3. Zusammenfassung

Einleitung

Programmanalyse:
zur *Kompilierzeit* Werte von Variablen und das dynamische Verhalten eines Programmes zur *Laufzeit* abschätzen
keine tatsächliche Ausführung des Programms: *statische Analyse*.
Haupteinsatzzwecke:
redundante und überflüssige Rechenschritte zu vermeiden (Compileroptimierung)
Verhalten von Software zu *validieren*

Der Satz von Rice:
„Jede nichttriviale Eigenschaft einer Turingmaschine, die sich auf die von der TM berechnete Funktion bezieht, ist unentscheidbar.“

$\rightarrow$ keine exakten Vorhersagen über das Verhalten von Programmen (nichttrivialen Programmteilen) möglich, höchstens *Abschätzungen*

Diese Abschätzung sollte **mindestens** alle Möglichkeiten, wie sich das Programm zur Laufzeit verhalten könnte beinhalten
→*Überapproximation*
eine Unterapproximation könnte möglicherweise ungewollte Ergebnisse nicht abdecken, hierzu ein kleines Beispiel:

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

S ist ein Statement, das y keinen anderen Wert mehr zuweist.
Man kann davon ausgehen, dass aus z := y folgt dass z nur die Werte 1 und 2 annehmen kann.
Ergebnis einer Programmanalyse:

- *„z kann nur den Wert* ***1*** *annehmen“* → deckt nicht alle Fälle ab
- *„z kann nur die Werte* ***1, 2 und 42*** *annehmen“* → akzeptabel, da einige Aussagen über das Ergebnis getroffen werden können

Natürlich ist eine Analyse, die nur genau alle richtigen Ergebnisse beinhaltet, noch besser.

Syntax

Es wird eine einfache, imperative Sprache namens WHILE verwendet:

- ein Programm besteht aus einem *Statement*, 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

$$a ::= x \mid n \mid a_1\ op_a\ a_2$$

$$b ::= \mathbf{true} \mid \mathbf{false} \mid \mathbf{not}\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_a\ a_2$$

$$S ::= [x := a]^l \mid [\texttt{skip}]^l \mid S_1\ \ S_2 \mid \texttt{if}\ [b]^l\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ [b]^l\ \texttt{do}\ S$$

Ein Beispiel in While:

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

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

berechnet die Fakultät von $\mathtt{x}$ und speichert sie in $\mathtt{z}$
Eindeutige Labels identifizieren einzelne Programmteile
so kann eine einfache Programmanalyse durchgeführt werden, die **Zuweisungsanalyse** (Reaching Definitions Analyses):

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
$[\mathtt{y} := \mathtt{x}]^1$ betritt $[\mathtt{z} := \mathtt{1}]^2$, nun werden Paare aus Variablen und den Labels, wo eine Zuweisung stattfindet, gebildet:
`(y,1)` und `(x,?)` erreichen das Label 2
(„?“ kennzeichnet dass die zugehörige Variable noch nicht initialisiert wurde)

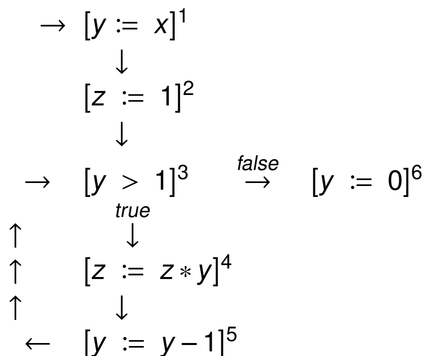
Alle Paare aus Variablen und Labels lassen sich für das Betreten $\mathrm{RD}_{entry}(l)$ und Verlassen $\mathrm{RD}_{exit}(l)$ jedes Blocks l in einer Tabelle darstellen:

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)`

Unterschied zwischen Unter- und Überapproximation: Im Label 5 könnte man `(z,2)` einfügen, in 6 aber nicht weglassen.

Datenflussanalyse

Die Datenflussanalyse kommt vor allem in Compilern zum Einsatz und lässt sich anwenden, solange sich ein Datenfluss-Graph zeichnen lässt.
Die Analyse basiert darauf, alle Möglichkeiten eines Flussdiagramms zu durchlaufen:



Ansatz mit Gleichungssystem

Aus dem Programm zur Fakultätsberechnung

$$[y := x]^1 \ [z := 1]^2 \ \texttt{while} \ [y > 1]^3$$

$$\text{do} \ ([z := z * y]^4 \ [y := y - 1]^5) \ [y := 0]^6$$

lassen sich folgende Gleichungen extrahieren:

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

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

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

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

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

Genauer lässt sich

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

schreiben, wobei zum Beispiel gilt:

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

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

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

$(P(\mathbf{Var}_* \times \mathbf{Lab}_*))^{12}$ lässt sich als Verband mit kleinstem Element auffassen:

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

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 fester Punkt in F ist und folglich $F^n(\overrightarrow{\emptyset})$ eine Lösung des obigen Gleichungssystems ist.

Ansatz mit Grenzwerten

Als Alternative zum Ansatz mit Gleichungssystemen lässt sich der Ansatz mit Grenzwerten verwenden, die Idee hierbei ist es, den Datenfluss nicht durch Gleichungen, sondern durch **Ungleichungen oder Grenzwerte** darzustellen.

Aus dem Programm zur Fakultätsberechnung

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

$$\mathrm{do}\ ([\mathtt{z} := \mathtt{z} * \mathtt{y}]^4;\ [\mathtt{y} := \mathtt{y} - 1]^5);\ [\mathtt{y} := 0]^6$$

lassen sich folgende Grenzwerte extrahieren:

$$
\begin{aligned}
RD_{exit}(1) &\supseteq RD_{entry}(1) \setminus \{(\mathtt{y}, l) \mid l \in \mathbf{Lab}\} \\
RD_{exit}(1) &\supseteq \{(\mathtt{y}, 1)\} \\
RD_{exit}(2) &\supseteq RD_{entry}(2) \setminus \{(\mathtt{z}, l) \mid l \in \mathbf{Lab}\} \\
RD_{exit}(2) &\supseteq \{(\mathtt{z}, 2)\} \\
RD_{exit}(3) &\supseteq RD_{entry}(3) \\
RD_{exit}(4) &\supseteq RD_{entry}(4) \setminus \{(\mathtt{z}, l) \mid l \in \mathbf{Lab}\} \\
RD_{exit}(4) &\supseteq \{(\mathtt{z}, 4)\} \\
RD_{exit}(5) &\supseteq RD_{entry}(5) \setminus \{(\mathtt{y}, l) \mid l \in \mathbf{Lab}\} \\
RD_{exit}(5) &\supseteq \{(\mathtt{y}, 5)\} \\
RD_{exit}(6) &\supseteq RD_{entry}(6) \setminus \{(\mathtt{y}, l) \mid l \in \mathbf{Lab}\} \\
RD_{exit}(6) &\supseteq \{(\mathtt{y}, 6)\}
\end{aligned}
$$

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
- 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:

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

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

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

aus, dass nicht bekannt ist, wo die nicht initialisierten Variablen definiert werden.

Schnell wird klar, dass die Lösung des Ansatzes mit Gleichungssystem 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

$$\begin{aligned} RD_{exit}(1) &\supseteq RD_{entry}(1) \backslash \{(y, l) \mid l \in \mathbf{Lab}\} \\ RD_{exit}(1) &\supseteq \{(y, 1)\} \end{aligned}$$

wird dann zum Beispiel

$$RD_{exit}(1) \supseteq RD_{entry}(1) \backslash \{(y, l) \mid l \in \mathbf{Lab}\} \cup \{(y, 1)\}$$

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

Grenzwertbasierte Analyse

Das Ziel der grenzwertbasierten Analyse ist es herauszufinden *„von welchem Block könnte der Kontrollfluss auf welchen übergehen?“*

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 deutlich schwerer als im vorherigen Beispiel zu sehen, wo hier der Kontrollfluss verläuft

Generell:

- drei Funktionen f, g und h
- f ist die Hauptfunktion
- g und h sind Parameter x für f
- x wird in f je der Parameter 1 übergeben, so dass das Ergebnis 7 sein wird
- ein Aufruf von f übergibt die Kontrolle an x 1
- dieser Aufruf von x übergibt die Kontrolle wiederum an den Funktionskörper von x

→ Problem: Man muss wissen, mit welchem Parameter f aufgerufen wurde, um zu wissen auf welches x der Kontrollfluss übergeht

Das Labeln einzelner Blöcke wie im imperativen WHILE wäre kompliziert, da die Blöcke geschachtelt auftreten
→ jeder Unterausdruck wird 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

Man befasst nun sich mit den Labels selbst, statt die Ein- und Austrittspunkte zu betrachten
Die Kontrollflussanalyse betrachtet nun Paare $(\hat{C},\hat{p})$ aus Funktionen:

- $\hat{C}(l)$ soll die Werte enthalten, die der Unterausdruck mit dem Label l annehmen könnte
- $\hat{p}(x)$ die Werte, die die Variable x annehmen könnte

Nun werden Grenzwerte gesammelt:

1. Abstrakten Funktionswerten Labels zuweisen:

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

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

2. Variablenwerte mit ihren Labels verbinden:

$$\hat{\rho}(\mathtt{x}) \subseteq \hat{\mathsf{C}}(1)$$

$$\hat{\rho}(\mathtt{y}) \subseteq \hat{\mathsf{C}}(3)$$

Die dritte und letzte Klasse von Grenzen befasst sich mit Funktionsaufrufen: Für jeden Punkt $[e_1\ e_2]^l$ und jede Funktion $[\mathtt{fn}\ x\ \texttt{=>}\ e]^{l'}$ 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.

3. Bedingte Grenzen:
Wenn die Funktion `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 `fn y => [y]`^{3} ist, sehen die bedingten Grenzen so aus:

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

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

Folgende Festlegungen von $\hat{\mathsf{C}}$ und $\hat{p}$ ergeben eine Lösung für obige Grenzwerte:

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

Zusammenfassend lässt sich sagen:

- Die Grenzwerteanalyse ist (wie der Name schon sagt) dem Ansatz mit Grenzwerten der Datenflussanalyse sehr ähnlich
- Der größte Unterschied zwischen beiden Analysen ist die deutlich komplexere Struktur dieser Grenzwertsysteme in der Grenzwertanalyse
- Das ist durch die Funktionsweise funktionaler Programmiersprachen notwendig

Abstrakte Interpretation

Hier wird 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.

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 Variablenzuweisungen im Programm zugeordnet, für das Programm zur Fakultätsberechnung

$$[y := x]^1;\ [z := 1]^2;\ \texttt{while}\ [y > 1]^3$$

$$\text{do}\ ([z := z * y]^4;\ [y := y - 1]^5);\ [y := 0]^6$$

ergibt sich folgende Trace:

$$((x,?),(y,?),(z,?),(y,1),(z,2),(z,4),(y,5),(z,4),(y,5),(y,6))$$

wenn die `while`-Schleife zweimal ausgeführt wird

Diese Traces enthalten genug Informationen, um eine Menge von semantischen Zuweisungen darzustellen:

$$
\begin{aligned}
CS_{exit}(1) &= \{tr : (y,1) \mid tr \in CS_{entry}(1)\} \\
CS_{exit}(2) &= \{tr : (z,2) \mid tr \in CS_{entry}(2)\} \\
CS_{exit}(3) &= CS_{entry}(3) \\
CS_{exit}(4) &= \{tr : (z,4) \mid tr \in CS_{entry}(4)\} \\
CS_{exit}(5) &= \{tr : (y,5) \mid tr \in CS_{entry}(5)\} \\
CS_{exit}(6) &= \{tr : (y,6) \mid tr \in CS_{entry}(6)\}
\end{aligned}
$$

$tr : (x, l)$ steht für die Aufnahme des Elements (x, l) zur Traceliste tr

Außerdem benötigt werden noch Gleichungen für den Kontrollfluss:

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

und die Tatsache dass Variablen am Anfang nicht initialisiert sind:

$$CS_{entry}(2) = \{((x,?),(y,?),(z,?))\}$$

Nun soll gezeigt werden, wie die Sammelsemantik genutzt werden kann, um die Analyse zu berechnen
G ist eine monotone Funktion folgender Art:

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

Hierzu verwendet man eine Funktionsabstraktion α und eine Konkretisierung γ, die in folgendem Zusammenhang stehen:

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

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

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

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

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

der zugehörige Teil in der Definition ist:

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

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

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

Die resultierende Gleichung

$$RD_{exit}(4) = (RD_{entry}(4) \backslash \{(z, l) \mid l \in \mathbf{Lab}\}) \cup \{(z, 4)\}$$

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

Typ- und Effektsysteme

Hier werden Programmen *Typen* zugewiesen, um Laufzeitfehler auszuschließen.

Die WHILE-Sprache wird um folgendes erweitert:
Ein Statement S stellt den Übergang von einem Zustand in einen anderen dar (wenn das Statement terminiert), es hat also den Typ $\Sigma \rightarrow \Sigma$, wobei Σ der Typ der Zustände ist

das kann folgendermaßen ausgedrückt werden:

$$S : \Sigma \rightarrow \Sigma$$

Ein Typ- und Effektystem besteht meist aus zwei Komponenten:

- Effektsystem
- Annotiertes Typsystem

Im **Effektsystem** kommen typischerweise Ausdrücke der Gestalt $S : \Sigma \stackrel{\varphi}{\rightarrow} \Sigma$ 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 $S : \Sigma_1 \rightarrow \Sigma_2$, wobei Σ_i die Eigenschaften von Zuständen (zum Beispiel ob eine Variable einen positiven Wert hat) beschreibt.

Annotierte Typsysteme

Annotierte Typsysteme lassen sich am Beispiel der WHILE-Sprache demonstrieren.
Hierzu wird wieder das Programm zur Fakultätsberechnung

$$[y := x]^1; [z := 1]^2; \texttt{while}\ [y > 1]^3$$

$$\text{do}\ ([z := z * y]^4; [y := y - 1]^5); [y := 0]^6$$

verwendet:

$$
\begin{array}{lll}
[\mathtt{y} := \mathtt{x}]^1 & : & \{(\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?)\} \rightarrow \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},?)\} \\
[\mathtt{z} := 1]^2 & : & \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},?)\} \rightarrow \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},2)\} \\
[\mathtt{y} > 1]^3 & : & \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},2)\} \rightarrow \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},2)\} \\
[\mathtt{z} := \mathtt{z} * \mathtt{y}]^4 & : & \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},2)\} \rightarrow \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{z},4)\} \\
[\mathtt{y} := \mathtt{y} - 1]^5 & : & \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{z},4)\} \rightarrow \{(\mathtt{x},?),(\mathtt{y},5),(\mathtt{z},4)\} \\
[\mathtt{y} := 0]^6 & : & \{(\mathtt{x},?),(\mathtt{y},5),(\mathtt{z},4)\} \rightarrow \{(\mathtt{x},?),(\mathtt{y},6),(\mathtt{z},2),(\mathtt{z},4)\}
\end{array}
$$

Also ergibt sich für das ganze Programm:

$$([\mathtt{y} := \mathtt{x}]^1;\ [\mathtt{z} := 1]^2;\ \mathtt{while}\ [\mathtt{y} > 1]^3$$

$$\text{do } ([\mathtt{z} := \mathtt{z} * \mathtt{y}]^4;\ [\mathtt{y} := \mathtt{y} - 1]^5);\ [\mathtt{y} := 0]^6) :$$

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

was wiederum dem Ergebnis in der Tabelle der Zuweisungsanalyse entspricht.

Effektsysteme

Die Vorgehensweise besteht hier darin, ein **Typsystem** wie im Abschnitt vorher mit **Informationen aus der Analyse** zu verbinden.
Ausdrücke folgen dann der Form

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

- Γ ist eine Typumgebung, die jeder Variable freien Variable von e Typen zuweist
- τ ist der Typ von e

Zur Vereinfachung wird hier davon ausgegangen, dass nur Basistypen wie `int` oder `bool` sowie Funktionstypen der Form $\tau_1 \rightarrow \tau_2$ existieren.

Call-Tracking Analysis: eine Analyse die Aufrufen nachgeht

Die Funktionstypen werden mit ihrem Effekt annotiert:

$$\texttt{int} \xrightarrow{\{X\}} \texttt{int}$$

bedeutet die Funktion:

- bildet Integer auf Integer ab
- verfügt über den Effekt {X}, d.h. es könnte während dem Aufruf die Funktion X aufgerufen werden

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 ist.

Transformationen

Ein wichtiges Aufgabengebiet der Programmanalyse:
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:

Dazu gehören zwei Vorgehensweisen:

- alle Variablen, die sich im Verlauf der Programmausführung nie ändern werden, durch Konstanten ersetzen
- Ausdrücke vereinfachen, indem sie teilweise ausgewertet werden

Wichtig ist hier: alle Unterausdrücke, die auf Variablen zugreifen, können schon vor der eigentlichen Ausführung ausgewertet werden

Um eine Transformation zu veranschaulichen soll folgendes simple Programm dienen:

$$[x := 10]^1; [y := x + 10]^2; [z := y + 10]^3$$

Eine Reichweitenanalyse ergibt folgendes:

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

Hieraus kann man folgende Transformationssequenz gewinnen:

$$
\begin{array}{lllll}
\text{RD} & \vdash & [x := 10]^1; & [y := x + 10]^2; & [z := y + 10]^3 \\
 & \triangleright & [x := 10]^1; & [y := 10 + 10]^2; & [z := y + 10]^3 \\
 & \triangleright & [x := 10]^1; & [y := 20]^2; & [z := y + 10]^3 \\
 & \triangleright & [x := 10]^1; & [y := 20]^2; & [z := 20 + 10]^3 \\
 & \triangleright & [x := 10]^1; & [y := 20]^2; & [z := 30]^3
\end{array}
$$

Zusammenfassung

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.

Vielen Dank für die Aufmerksamkeit!