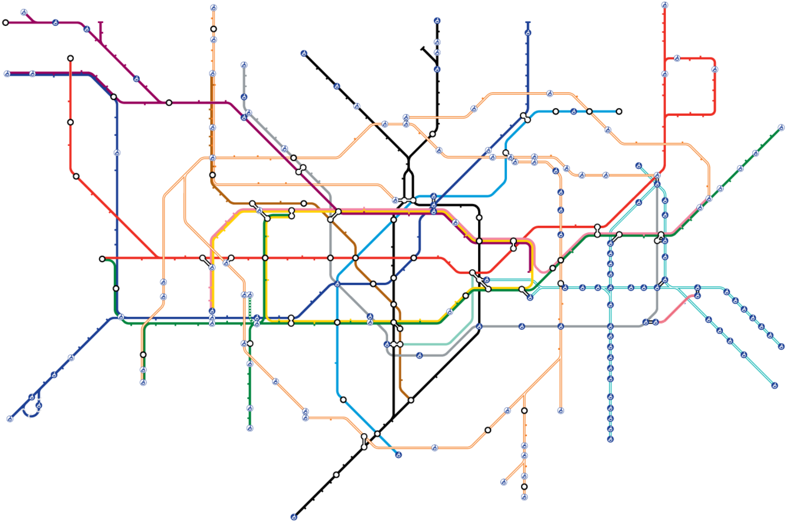
Teil 3.3: Statische Analyse

Falk Howar

Softwarekonstruktion WS 2018/19

LS14

Statische Analyse



Wäre es nicht super,
wenn man Coding Standards automatisiert prüfen könnte?

Was könnte man noch prüfen?

Statische Analyse

- Analyse beim Kompilieren
- Basiert auf Modell, AST oder CFG
- Prüft Eigenschaften ohne Code auszuführen
- Nutzt oft **Über-Approximierung** von möglichem Verhalten des Codes

Wdh. Getypte Ausdrücke

$a \in \mathbf{AExp}$ (Arithmetische Ausdrücke)

$a ::= \; x \mid \mathtt{n} \mid (a_1 \; op_a \; a_2)$

$b \in \mathbf{BExp}$ (Boolesche Ausdrücke)

$b ::= \; y \mid \mathtt{true} \mid \mathtt{false} \mid \neg\, b \mid (b_1 \; op_b \; b_2) \mid (a_1 \; op_r \; a_2)$

x Variable Typ `Nat`

y Variable Typ `Boolean`

$\mathtt{n}$ Literal für $n \in \mathbb{Z}$

$op_a \in \mathbf{Op_a}$ arithmetische Op.: $+,\ -$

$op_b \in \mathbf{Op_b}$ Boolesche Op.: $\wedge,\ \vee$

$op_r \in \mathbf{Op_r}$ Relationale Op.: $>,\ <,\ =$

Die While Sprache

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

x Variable Typ `Nat` oder `Boolean`

a Ausdruck von passendem Typ zu x

b Ausdruck aus $\mathbf{BExp}$

$S \in \mathbf{Stmt}$ Anweisungen

$l \in \mathbf{Lab}$ Label

Annahme: Label identifizieren Anweisungen eindeutig!

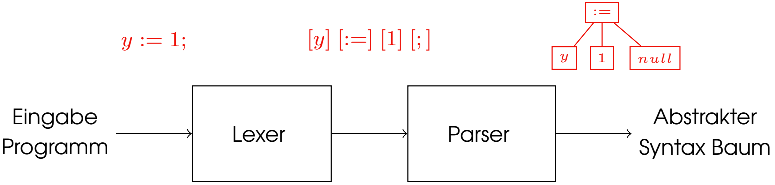
Beispiel

While Program:

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

Parsen / Übersetzen von While

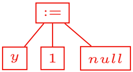
Compiler Front End:



Abstrakter Syntax Baum (AST)

Intuitiv: Ein abstrakter Syntax Baum ist ein Baum, der die syntaktische Struktur einen Programms ausdrückt

- Ausgabe eines Compiler Front-end
- Grundlage für für Generierung von Code



- Dient auch als Grundlage für einfache statische Code-Analyse.
- Z.B.: **Prüfe, dass alle Namen von Klassen mit Großbuchstaben beginnen**

AST Beispiel

While Programm:

$[y := x]^1;$

$[z := 1]^2;$

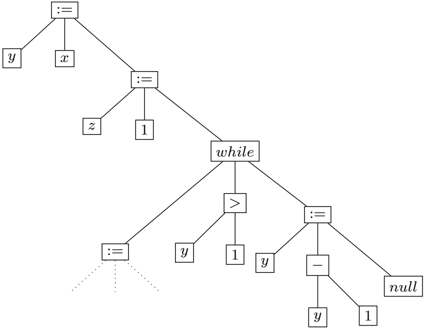
`while` $[y > 1]^3$ `do`

$\quad [z := x * y]^4;$

$\quad [y := y - 1]^5$

$[y := 0]^6;$

Abstrakter Syntax Baum:



Kontrollflussgraph (CFG)

Intuitiv:
Der Kontrollflussgraph eines Programms P ist ein Graph (V, E) mit

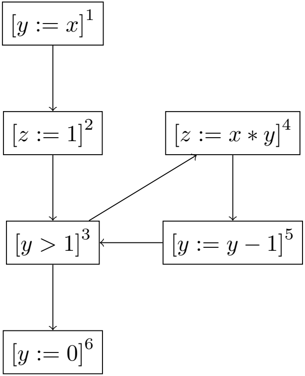
- Menge Knoten V, die einen Knoten für jede Anweisung in P enthält und
- Menge Kanten $E \subseteq V \times V$, die eine Kante $(v, v') \in E$ enthält wenn P von v zu v' übergehen kann.

CFG Beispiel

While Programm:

$$
\begin{array}{l}
[y := x]^1; \\
[z := 1]^2; \\
\texttt{while}\ [y > 1]^3\ \texttt{do} \\
\quad [z := x * y]^4; \\
\quad [y := y - 1]^5 \\
[y := 0]^6;
\end{array}
$$

CFG:



Formalisierung Kontrollfluss (1)

Elementare Blöcke eines Programms: $blocks : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Blocks})$

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

Labels eines Programms: $labels : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab})$

$$labels(S) = \{\ l \in \mathbf{Lab}\ :\ [B]^l \in blocks(S)\ \}$$

$\mathbf{Blocks}$: elementare Blöcke der Form $[x := a]^l$, $[\texttt{skip}]^l$ bzw. $[b]^l$

Formalisierung Kontrollfluss (2)

Start-Knoten: $init : \mathbf{Stmt} \to \mathbf{Lab}$

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

Formalisierung Kontrollfluss (2)

Start-Knoten: $init : \mathbf{Stmt} \to \mathbf{Lab}$

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

End-Knoten: $final : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab})$

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

Formalisierung Kontrollfluss (3)

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

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

Umgekehrter Kontrollfluss: $flow^R : \mathbf{Stmt} \rightarrow \mathcal{P}(\mathbf{Lab} \times \mathbf{Lab})$

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

Formalisierung Kontrollfluss (4)

Kontrollflussgraph (CFG)

Der Kontrollflussgraph eines $While$ Programms $S_\star$ ist ein Graph (V, E) mit

- Knoten $V = labels(S_\star)$ und
- Kanten $E = flow(S_\star)$.

Annahmen: (vereinfachen Präsentation einiger Analysen)

- Isolierter Start-Knoten: $\forall l \in labels(S_\star) \;.\; (l, init(S_\star)) \notin flow(S_\star)$
- Isolierter End-Knoten: $\forall l \in labels(S_\star) \;.\; (final(S_\star), l) \notin flow(S_\star)$

(Kann durch Hinzufügen von `skip` Anweisungen erreicht werden)

Einschub: Cyclomatische Komplexität

Cyclomatische Komplexität (CC)

Cyclomatische Komplexität (CC)

Die Cyclomatische Komplexität eines Programms P entspricht der Anzahl der unabhängigen Pfade in P. Sei e die Zahl der Kanten im Kontrollflussgraph von P, n die Zahl der Knoten im Kontrollflussgraph und p die Zahl der Zusammenhangskomponenten. Dann gilt

$$CC = e - n + 2p$$

- Komplexität wird gemessen durch Anzahl der Pfade
- Viele Pfade $\Leftrightarrow$ hohe Komplexität
- $p = 1$ für Programme mit einem Einstiegspunkt.

Cyclomatische Komplexität: Beispiel

While Programm:

$[y := x]^1;$

$[z := 1]^2;$

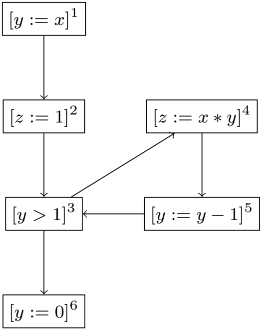
`while` $[y > 1]^3$ `do`

$\quad [z := x * y]^4;$

$\quad [y := y - 1]^5$

$[y := 0]^6;$

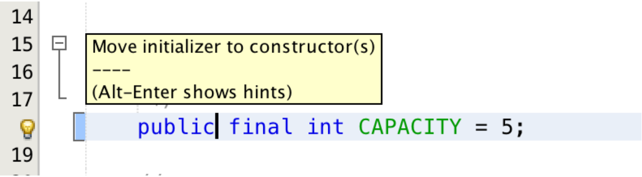
Kontrollflussgraph:



$$CC = 6 - 6 + 2 = 2$$

Weitere Analysen ...

AST-basierte Analyse



AST-basierte Analyse

AST-basierte statische Code-Analyse

AST-basierte statische Code-Analyse sucht nach Merkmalen / Sttruktur im abstrakten Syntax Baum eines Programms.

Beispiele:

- Initialisierung von Variablen
- Benennung von Klassen
- ...

AST-basierte Analyse: Beispiel

While Programm:

$[y := x]^1;$

$[z := 1]^2;$

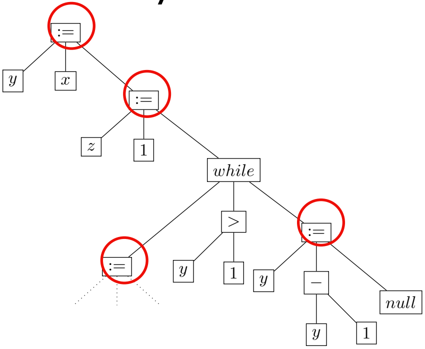
`while` $[y > 1]^3$ `do`

$[z := x * y]^4;$

$[y := y - 1]^5$

$[y := 0]^6;$

Abstrakter Syntax Baum:



Länge von Variablennamen:

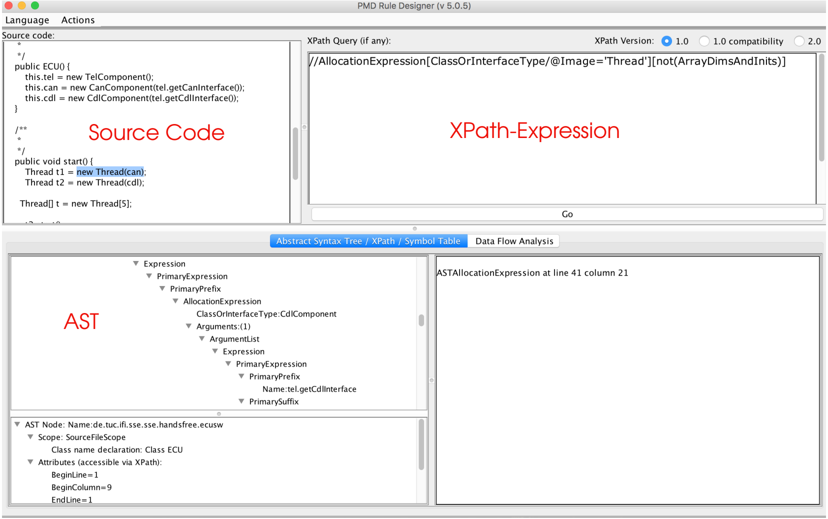
$(:=)$-Knoten, an denen erstes Kind ein zu kurzes Label (z.B. < 3) hat

PMD

- Bibliothek für statische Analyse
- Viele unterstützte Sprachen: C, MatLab, Java, ...
- IDE Integration
- Maven Integration

`https://pmd.github.io/`

PMD Rule Designer



Statische Analyse von Modellen

Natürlich funktionert das auch für Modelle!

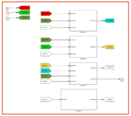
Beispiel: MXAM

MXAM: Model Examiner

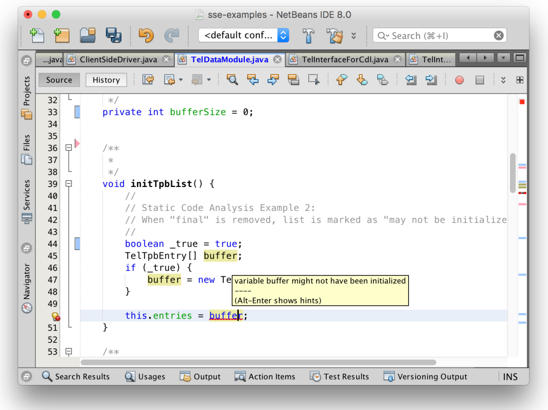
- Eingesetzt bei modell-basierter Entwicklung

ISO26262-6 / Table 1 – Topics to be covered by modeling guidelines	ASIL A	ASIL B	ASIL C	ASIL D
1a) Enforcement of low complexity	++	++	++	++
1b) Use of language subsets	++	++	++	++
1c) Enforcement of strong typing	++	++	++	++
1d) Use of defensive implementation techniques	o	+	++	++
1e) Use of established design principles	+	+	+	++
1f) Use of unambiguous graphical representation	+	++	++	++
1g) Use of style guides	+	++	++	++
1h) Use of naming conventions	++	++	++	++

ISO26262-6 / Table 8 – Design principles	ASIL A	ASIL B	ASIL C	ASIL D
1a) One entry and one exit point in subprograms and functions	++	++	++	++
1b) No dynamic objects or variables, or else online test during their creation	+	++	++	++
1c) Initialization of variables	++	++	++	++
1d) No multiple use of variable names	+	++	++	++
1e) Avoid global variables or else justify their usage	+	+	++	++
1f) Limited use of pointers	o	+	+	++
1g) No implicit type conversion	+	++	++	++
1h) No hidden data flow or control flow	+	++	++	++
1i) No unconditional jumps	++	++	++	++
1j) No recursions	+	+	++	++



Datenfluss-Analyse



- Manchmal reicht der AST nicht aus . . .
- Analyse auf Basis des Kontrollflussgraphen (CFG)

Nicht initialisierte Variable in While Programm

While Programm:

$$
\begin{array}{l}
[y := 5]^1; \\
[z := 1]^2; \\
\texttt{while } [y > 1]^3 \texttt{ do} \\
\quad [z := x * y]^4; \\
\quad [y := y - 1]^5 \\
[y := 0]^6;
\end{array}
$$

Problem:

x nicht initialisiert!

Nicht initialisierte Variable in While Programm

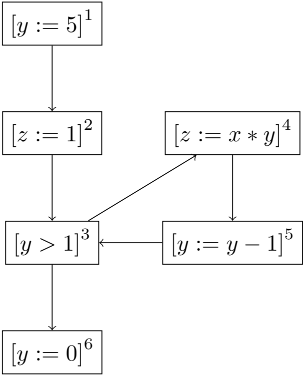
While Programm:

$$
\begin{array}{l}
[y := 5]^1; \\
[z := 1]^2; \\
\texttt{while}\ [y > 1]^3\ \texttt{do} \\
\quad [z := x * y]^4; \\
\quad [y := y - 1]^5 \\
[y := 0]^6;
\end{array}
$$

Problem:

x nicht initialisiert!

CFG:



Nicht initialisierte Variable in While Programm

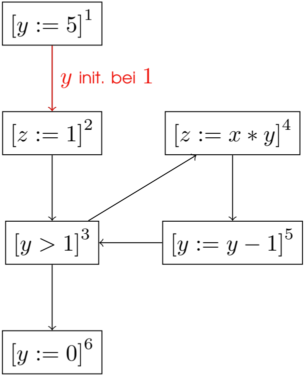
While Programm:

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

Problem:

x nicht initialisiert!

CFG:



Nicht initialisierte Variable in While Programm

While Programm:

$[y := 5]^1;$

$[z := 1]^2;$

$\texttt{while}\ [y > 1]^3\ \texttt{do}$

$\quad [z := x * y]^4;$

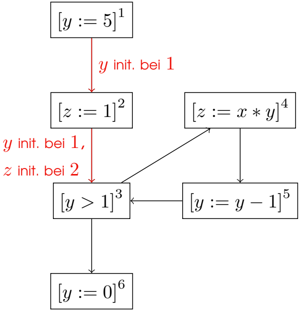
$\quad [y := y - 1]^5$

$[y := 0]^6;$

Problem:

x nicht initialisiert!

CFG:



Reaching Definitions

Reaching Definitions

Reaching Definitions (RD) berechnet, welche Zuweisungen zu Variablen zu welchen Anweisungen im Programm propagiert werden können.

Anwendung: z.B. Entdecken potenziell uninitialisierter Variablen

Reaching Definitions

Reaching Definitions

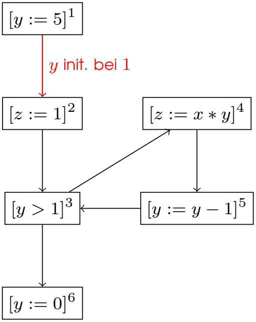
Reaching Definitions (RD) berechnet, welche Zuweisungen zu Variablen zu welchen Anweisungen im Programm propagiert werden können.

Anwendung: z.B. Entdecken potenziell uninitialisierter Variablen

Idee:

Über-approximieren des Datenfluss auf Basis des CFG.

- Über-approximieren: Wir ignorieren Entscheidungen bei Verzweigungen
- Datenfluss: Wir simulieren den Fluss von Information (z.B. Zuweisungen) entlang von Konaten des CFG



RD klassisches Beispiel für Datenfluss-Analyse

Reaching Definitions: Ansatz

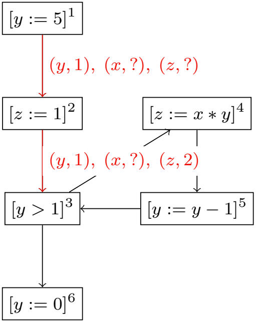
- System von Gleichungen drückt Bedingungen über Datenfluss im CFG aus
- Wir suchen die "kleinste" Lösung, die Bedingungen erfüllt

Definitionen:

Sei (x, l) aus $\mathbf{Var} \times (\mathbf{Lab} \cup \{?\})$.

- Paar (x, l) bedeutet: Anweisung l ändert x
- $(x, ?)$ bedeutet: x ist nicht initialisiert

- Fluss in l: $\mathrm{RD}_{entry}(l) \subseteq \mathbf{Var} \times (\mathbf{Lab} \cup \{?\})$
- Fluss aus l: $\mathrm{RD}_{exit}(l) \subseteq \mathbf{Var} \times (\mathbf{Lab} \cup \{?\})$



Reaching Definitions: Formalisierung (1)

Kill- und Gen-Funktionen:

$$
\begin{aligned}
kill_{\mathrm{RD}}(\ [x := a]^l\ ) &= \{(x, ?)\} \\
&\cup\ \{\ (x, l')\ :\ [B]^{l'} \text{ ist Zuweisung zu } x \text{ in } S_\star\} \\
kill_{\mathrm{RD}}(\ [\texttt{skip}]^l\ ) &= \emptyset \\
kill_{\mathrm{RD}}(\ [b]^l\ ) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
gen_{\mathrm{RD}}(\ [x := a]^l\ ) &= \{(x, l)\} \\
gen_{\mathrm{RD}}(\ [\texttt{skip}]^l\ ) &= \emptyset \\
gen_{\mathrm{RD}}(\ [b]^l\ ) &= \emptyset
\end{aligned}
$$

Reaching Definitions: Formalisierung (2)

Datenfluss Gleichungen: (Zu verstehen als Bedingungen an Datenfluss)

$$\mathrm{RD}_{entry}(l) = \begin{cases} \{\ (x, ?)\ :\ x \in FV(S_\star)\ \} & \text{für } l = init(S_\star) \\ \bigcup \{\ \mathrm{RD}_{exit}(l')\ :\ (l', l) \in flow(S_\star)\} & \text{sonst} \end{cases}$$

$$\mathrm{RD}_{exit}(l) = (\mathrm{RD}_{entry}(l)\ \setminus\ kill_{\mathrm{RD}}([B]^l))\ \cup\ gen_{\mathrm{RD}}([B]^l)$$
$$\text{für } [B]^l \in blocks(S_\star)$$

Reaching Definitions: Formalisierung (2)

Datenfluss Gleichungen: (Zu verstehen als Bedingungen an Datenfluss)

$$\mathrm{RD}_{entry}(l) = \begin{cases} \{\ (x,?)\ :\ x \in FV(S_\star)\ \} & \text{für } l = init(S_\star) \\ \bigcup \{\ \mathrm{RD}_{exit}(l')\ :\ (l',l) \in flow(S_\star)\} & \text{sonst} \end{cases}$$

$$\mathrm{RD}_{exit}(l) = (\mathrm{RD}_{entry}(l)\ \setminus\ kill_{\mathrm{RD}}([B]^l))\ \cup\ gen_{\mathrm{RD}}([B]^l)$$
$$\text{für } [B]^l \in blocks(S_\star)$$

For $\mathbf{Lab}_\star = \{1, \ldots, k\}$

$$\mathrm{RD}(S_\star)\ =\ \big(\mathrm{RD}_{entry}(1),\ \mathrm{RD}_{exit}(1),\ \ldots,\ \mathrm{RD}_{entry}(k),\ \mathrm{RD}_{exit}(k)\ \big)$$

$\mathbf{Lab}_\star = labels(S_\star)$ und

$FV(S)$: Variablen in Anweisungen und Ausdrücken von S

Reaching Definitions: Einfacher Algorithmus

Gesucht:
Funktionen $reach$ mit $reach_{entry}, reach_{exit} : \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{Var} \times (\mathbf{Lab} \cup \{?\}))$ die $\mathrm{RD}(S_\star)$ erfüllen.

Superscripts zählen Schritte des Algorithmus.

Reaching Definitions: Einfacher Algorithmus

Gesucht:

Funktionen $reach$ mit $reach_{entry}, reach_{exit} : \mathbf{Lab}_\star \rightarrow \mathcal{P}(\mathbf{Var} \times (\mathbf{Lab} \cup \{?\}))$ die $\mathrm{RD}(S_\star)$ erfüllen.

Initialisierung:

- $\mathrm{reach}^0_{entry}(l) = \{(x, ?) \ : \ x \in \mathbf{Var}_\star\}$ für $l = init(S_\star)$
- $\mathrm{reach}^0_{entry}(l) = \emptyset$ sonst

Superscripts zählen Schritte des Algorithmus.

Reaching Definitions: Einfacher Algorithmus

Gesucht:

Funktionen $reach$ mit $reach_{entry}, reach_{exit} : \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{Var} \times (\mathbf{Lab} \cup \{?\}))$ die $\mathrm{RD}(S_\star)$ erfüllen.

Initialisierung:

- $\mathrm{reach}^0_{entry}(l) = \{(x, ?) \ : \ x \in \mathbf{Var}_\star\}$ für $l = init(S_\star)$
- $\mathrm{reach}^0_{entry}(l) = \emptyset$ sonst

Vervollständigen:

$$\mathrm{reach}^i_{exit}(l) = \begin{cases} \mathrm{reach}^i_{entry}(l) \setminus \{(x, l') \ : \ l' \in \mathbf{Lab}\}) \cup \{(x, l)\} & \text{für } [x := a]^l \\ \mathrm{reach}^i_{entry}(l) & \text{sonst} \end{cases}$$

Superscripts zählen Schritte des Algorithmus.

Reaching Definitions: Einfacher Algorithmus

Gesucht:

Funktionen $reach$ mit $reach_{entry}, reach_{exit} : \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{Var} \times (\mathbf{Lab} \cup \{?\}))$ die $\mathrm{RD}(S_\star)$ erfüllen.

Initialisierung:

- $\mathrm{reach}^0_{entry}(l) = \{(x, ?) \ : \ x \in \mathbf{Var}_\star\}$ für $l = init(S_\star)$
- $\mathrm{reach}^0_{entry}(l) = \emptyset$ sonst

Vervollständigen:

$$\mathrm{reach}^i_{exit}(l) = \begin{cases} \mathrm{reach}^i_{entry}(l) \setminus \{(x, l') \ : \ l' \in \mathbf{Lab}\}) \cup \{(x, l)\} & \text{für } [x := a]^l \\ \mathrm{reach}^i_{entry}(l) & \text{sonst} \end{cases}$$

Solange unstabil:

$$\mathrm{reach}^{i+1}_{entry}(l) = \bigcup_{\{l' \in \mathbf{Lab}_\star \ : \ (l', l) \in flow(S_\star)\}} \mathrm{reach}^i_{exit}(l')$$

Superscripts zählen Schritte des Algorithmus.

Reaching Definitions: Terminierung

Algorithmus terminiert wenn

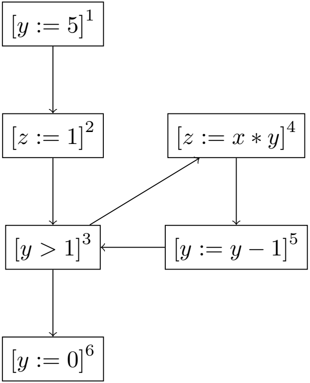
$$reach^{i+1} = reach^{i}$$

Punkt existiert:

- Nur endlich viele Paare
- Neue Paare (x, l) werden nur einmal hinzugefügt
- Alle Mengen wachsen monoton.

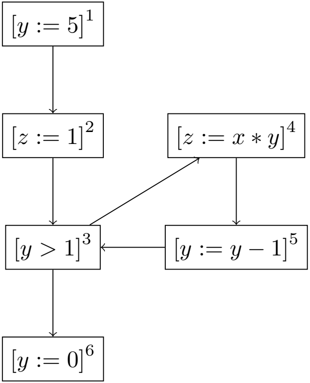
"Kleinste" gültige Lösung von RD

Reaching Definitions: Beispiel



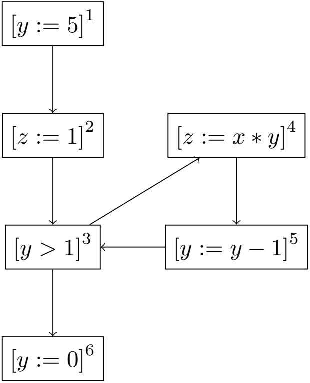
l	$reach_{entry}(l)$	$reach_{exit}(l))$
1	$(x, ?)$, $(y, ?)$, $(z, ?)$	
2		
3		
4		
5		
6		

Reaching Definitions: Beispiel



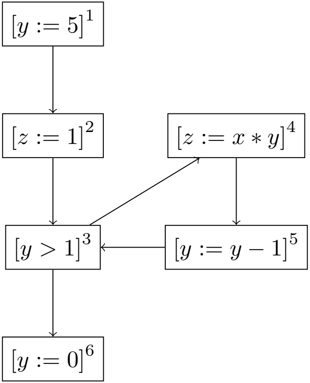
l	$reach_{entry}(l)$	$reach_{exit}(l))$
1	$(x,?), (y,?), (z,?)$	$(x,?), (y,1), (z,?)$
2		$(z,2)$
3		
4		$(z,4)$
5		$(y,5)$
6		$(y,6)$

Reaching Definitions: Beispiel



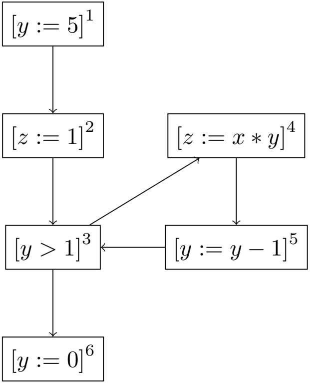
l	$reach_{entry}(l)$	$reach_{exit}(l))$
1	$(x, ?)$, $(y, ?)$, $(z, ?)$	$(x, ?)$, $(y, 1)$, $(z, ?)$
2	$(x, ?)$, $(y, 1)$, $(z, ?)$	$(z, 2)$
3	$(z, 2)$, $(y, 5)$	
4		$(z, 4)$
5	$(z, 4)$	$(y, 5)$
6		$(y, 6)$

Reaching Definitions: Beispiel



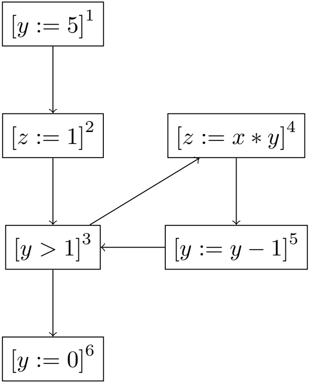
l	$reach_{entry}(l)$	$reach_{exit}(l))$
1	$(x,?)$, $(y,?)$, $(z,?)$	$(x,?)$, $(y,1)$, $(z,?)$
2	$(x,?)$, $(y,1)$, $(z,?)$	$(x,?)$, $(y,1)$, $(z,2)$
3	$(z,2)$, $(y,5)$	$(z,2)$, $(y,5)$
4		$(z,4)$
5	$(z,4)$	$(y,5)$, $(z,4)$
6		$(y,6)$

Reaching Definitions: Beispiel



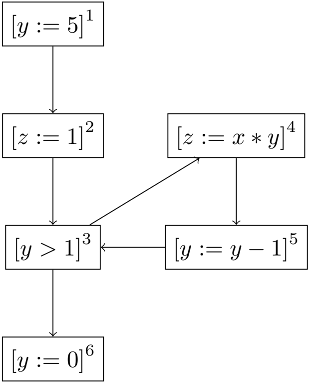
l	$reach_{entry}(l)$	$reach_{exit}(l))$
1	$(x,?), (y,?), (z,?)$	$(x,?), (y,1), (z,?)$
2	$(x,?), (y,1), (z,?)$	$(x,?), (y,1), (z,2)$
3	$(x,?), (y,1), (z,2)$, $(y,5)$, $(z,4)$	$(z,2), (y,5)$
4	$(z,2), (y,5)$	$(z,4)$
5	$(z,4)$	$(y,5)$, $(z,4)$
6	$(z,2)$, $(y,5)$,	$(y,6)$

Reaching Definitions: Beispiel



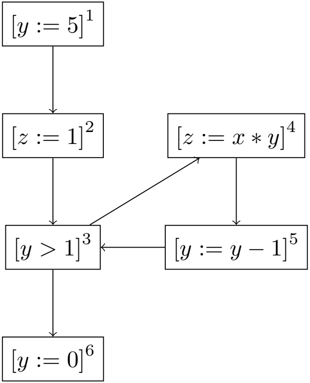
l	$reach_{entry}(l)$	$reach_{exit}(l))$
1	$(x, ?)$, $(y, ?)$, $(z, ?)$	$(x, ?)$, $(y, 1)$, $(z, ?)$
2	$(x, ?)$, $(y, 1)$, $(z, ?)$	$(x, ?)$, $(y, 1)$, $(z, 2)$
3	$(x, ?)$, $(y, 1)$, $(z, 2)$, $(y, 5)$, $(z, 4)$	$(x, ?)$, $(y, 1)$, $(z, 2)$, $(y, 5)$, $(z, 4)$
4	$(z, 2)$, $(y, 5)$	$(z, 4)$, $(y, 5)$
5	$(z, 4)$, $(y, 5)$	$(y, 5)$, $(z, 4)$
6	$(z, 2)$, $(y, 5)$,	$(y, 6)$, $(z, 2)$

Reaching Definitions: Beispiel



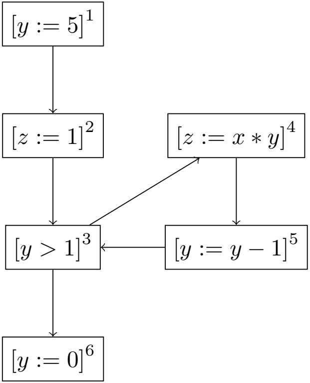
l	$reach_{entry}(l)$	$reach_{exit}(l))$
1	$(x,?)$, $(y,?)$, $(z,?)$	$(x,?)$, $(y,1)$, $(z,?)$
2	$(x,?)$, $(y,1)$, $(z,?)$	$(x,?)$, $(y,1)$, $(z,2)$
3	$(x,?)$, $(y,1)$, $(z,2)$, $(y,5)$, $(z,4)$	$(x,?)$, $(y,1)$, $(z,2)$, $(y,5)$, $(z,4)$
4	$(x,?)$, $(y,1)$, $(z,2)$, $(y,5)$, $(z,4)$	$(z,4)$, $(y,5)$
5	$(z,4)$, $(y,5)$	$(y,5)$, $(z,4)$
6	$(x,?)$, $(y,1)$, $(z,2)$, $(y,5)$, $(z,4)$	$(y,6)$, $(z,2)$

Reaching Definitions: Beispiel



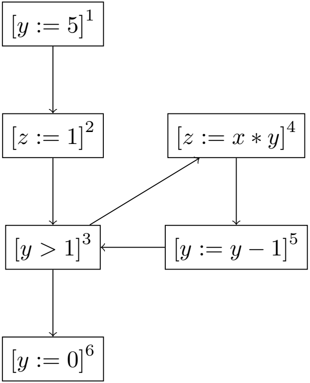
l	$reach_{entry}(l)$	$reach_{exit}(l))$
1	$(x,?), (y,?), (z,?)$	$(x,?), (y,1), (z,?)$
2	$(x,?), (y,1), (z,?)$	$(x,?), (y,1), (z,2)$
3	$(x,?), (y,1), (z,2), (y,5), (z,4)$	$(x,?), (y,1), (z,2), (y,5), (z,4)$
4	$(x,?), (y,1), (z,2), (y,5), (z,4)$	$(x,?), (y,1), (z,4), (y,5)$
5	$(z,4), (y,5)$	$(y,5), (z,4)$
6	$(x,?), (y,1), (z,2), (y,5), (z,4)$	$(x,?), (y,6), (z,2), (z,4)$

Reaching Definitions: Beispiel



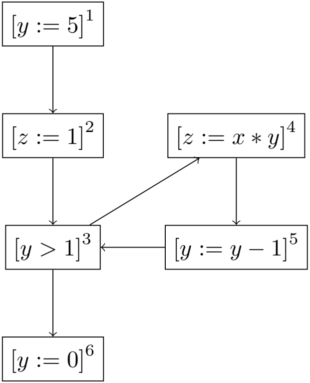
l	$reach_{entry}(l)$	$reach_{exit}(l))$
1	$(x,?)$, $(y,?)$, $(z,?)$	$(x,?)$, $(y,1)$, $(z,?)$
2	$(x,?)$, $(y,1)$, $(z,?)$	$(x,?)$, $(y,1)$, $(z,2)$
3	$(x,?)$, $(y,1)$, $(z,2)$, $(y,5)$, $(z,4)$	$(x,?)$, $(y,1)$, $(z,2)$, $(y,5)$, $(z,4)$
4	$(x,?)$, $(y,1)$, $(z,2)$, $(y,5)$, $(z,4)$	$(x,?)$, $(y,1)$, $(z,4)$, $(y,5)$
5	$(x,?)$, $(y,1)$, $(z,4)$, $(y,5)$	$(y,5)$, $(z,4)$
6	$(x,?)$, $(y,1)$, $(z,2)$, $(y,5)$, $(z,4)$	$(x,?)$, $(y,6)$, $(z,2)$, $(z,4)$

Reaching Definitions: Beispiel



l	$reach_{entry}(l)$	$reach_{exit}(l))$
1	$(x,?), (y,?), (z,?)$	$(x,?), (y,1), (z,?)$
2	$(x,?), (y,1), (z,?)$	$(x,?), (y,1), (z,2)$
3	$(x,?), (y,1), (z,2), (y,5), (z,4)$	$(x,?), (y,1), (z,2), (y,5), (z,4)$
4	$(x,?), (y,1), (z,2), (y,5), (z,4)$	$(x,?), (y,1), (z,4), (y,5)$
5	$(x,?), (y,1), (z,4), (y,5)$	$(x,?), (y,5), (z,4)$
6	$(x,?), (y,1), (z,2), (y,5), (z,4)$	$(x,?), (y,6), (z,2), (z,4)$

Reaching Definitions: Beispiel

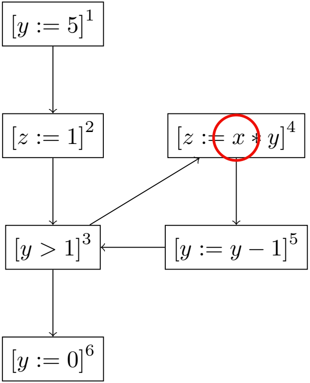


l	$reach_{entry}(l)$	$reach_{exit}(l))$
1	$(x,?), (y,?), (z,?)$	$(x,?), (y,1), (z,?)$
2	$(x,?), (y,1), (z,?)$	$(x,?), (y,1), (z,2)$
3	$(x,?), (y,1), (z,2), (y,5), (z,4)$	$(x,?), (y,1), (z,2), (y,5), (z,4)$
4	$(x,?), (y,1), (z,2), (y,5), (z,4)$	$(x,?), (y,1), (z,4), (y,5)$
5	$(x,?), (y,1), (z,4), (y,5)$	$(x,?), (y,5), (z,4)$
6	$(x,?), (y,1), (z,2), (y,5), (z,4)$	$(x,?), (y,6), (z,2), (z,4)$

$$\mathrm{RD}_{entry}(l) = \begin{cases} \{\, (x,?) \;:\; x \in FV(S_\star) \,\} & \text{für } l = init(S_\star) \\ \bigcup \{\, \mathrm{RD}_{exit}(l') \;:\; (l', l) \in flow(S_\star) \} & \text{sonst} \end{cases}$$

$$\mathrm{RD}_{exit}(l) = (\mathrm{RD}_{entry}(l) \setminus kill_{\mathrm{RD}}([B]^l)) \cup gen_{\mathrm{RD}}([B]^l)$$

Reaching Definitions: Beispiel



l	$reach_{entry}(l)$	$reach_{exit}(l))$
1	$(x,?), (y,?), (z,?)$	$(x,?), (y,1), (z,?)$
2	$(x,?), (y,1), (z,?)$	$(x,?), (y,1), (z,2)$
3	$(x,?), (y,1), (z,2), (y,5), (z,4)$	$(x,?), (y,1), (z,2), (y,5), (z,4)$
4	$(x,?), (y,1), (z,2), (y,5), (z,4)$	$(x,?), (y,1), (z,4), (y,5)$
5	$(x,?), (y,1), (z,4), (y,5)$	$(x,?), (y,5), (z,4)$
6	$(x,?), (y,1), (z,2), (y,5), (z,4)$	$(x,?), (y,6), (z,2), (z,4)$

$$\mathrm{RD}_{entry}(l) = \begin{cases} \{\, (x,?) \;:\; x \in FV(S_\star) \,\} & \text{für } l = init(S_\star) \\ \bigcup \{\, \mathrm{RD}_{exit}(l') \;:\; (l', l) \in flow(S_\star) \} & \text{sonst} \end{cases}$$

$$\mathrm{RD}_{exit}(l) = (\mathrm{RD}_{entry}(l) \setminus kill_{\mathrm{RD}}([B]^l)) \cup gen_{\mathrm{RD}}([B]^l)$$

Potential use of uninitialized variable!

Reaching Definitions: Korrektheit

Zu zeigen:
Wenn in einer Ausführung von $S_\star$, die Zuweisung zu x von Anweisung l bis zu Anweisung l' überlebt, dann ist (x, l) in $reach_{entry}(l')$ von jeder gültigen Lösung $reach$ für $\mathrm{RD}(S_\star)$.

Wir haben noch gar nicht definiert, wie While ausgeführt wird (nächster Block)

Beweis: Andere Vorlesungen (z.B. FMS)

Weitere Analysen

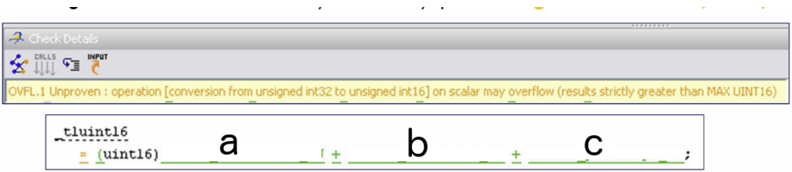
- Available Expressions

  Welche Ausdrücke werden wann berechnet und werden an welchen anderen Stellen im Programm noch benötigt

- Live Variables

  Welche Variablen werden zu welchem Zeitpunkt noch benötigt

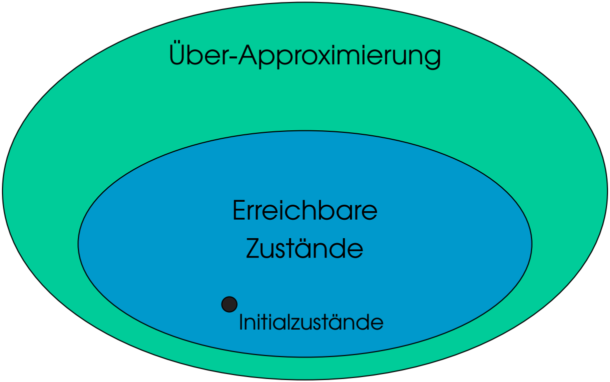
Abstrakte Interpretation



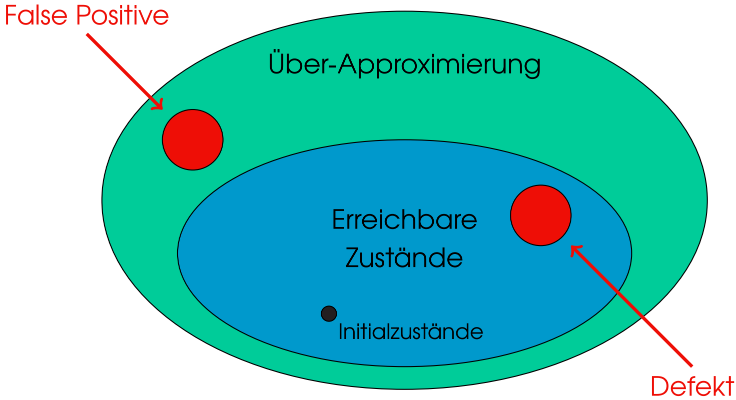
- Datenfluss Analyse
- Propagierung von abstrakten Werten über den CFG
- Abstrakte Domänen, z.B. Intervalle

Bild: Polyspace Overflow Warnung

‘False Positives’ bei statischer Analyse mit Über-Approximierung



‘False Positives’ bei statischer Analyse mit Über-Approximierung



Datenfluss-Analyse: Demonstration

...

Zusammenfassung Statische Analyse (1)

- Compile-Zeit Analyse
- Analyse auf Basis von AST oder CFG
- Code wird nicht ausgeführt

Zusammenfassung Statische Analyse (2)

AST-basiert:

- Suche nach Mustern im abstrakten Syntax Baum
- Einfache Eigenschaften (z.B. Benennung)

CFG-basiert:

- Komplexere Code Metriken (z.B. CC)
- Datenfluss Analyse
- Über-Approximierung von Datenfluss

Statische Modell Analyse:

- Analog zu Code