Programm Analyse

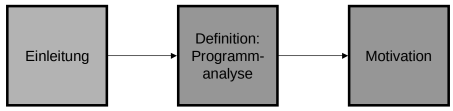
Dr.-Ing. Florian Martin
www.cs.uni-sb.de/~martin
Lehrstuhl Prof. Dr. R. Wilhelm -- Fachrichtung Informatik

Weiterführende Literatur:

Nielson, Nielson, Hankin: Principles of Program Analysis

Springer 1998

Überblick



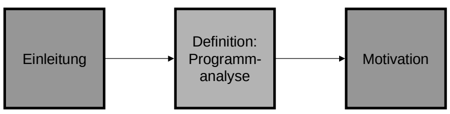
Einordnung des Vortrags

Themengebiete:
- Compilerbau
- Semantik
- Optimierung

Gliederung

- Definition, Beispiele
- Techniken der Programmanalyse
- Theorie der Programmanalyse
- Generierung von Programmanalysatoren

Überblick



Ziele der Programmanalyse

Berechnen (mechanisch) von Informationen über ein Programm um
- zu prüfen, ob Programmoptimierungen zulässig sind
  = ersetzen eines Programms durch ein „besseres" Programm, daß das „gleiche" tut
- produzieren von „intelligenten" Fehlermeldungen
- prüfen von anderen Aspekten des Programms (z.B. Zeitverhalten)

Entfernen von Überflüssigen Zuweisungen

```
x := 3;
a := z;
x := a;
```

```
a := z;
x := a;
```


Herausziehen von Schleifeninvariantem Code

```
for (i=0;i<10;i++) {
  z = r*r;
  k = k*i;
}
```

```
z = r*r;
for (i=0;i<10;i++) {
  k = k*i;
}
```


Benutzung von Uninitialisierten Variablen

```
int f(int x) {
int a;

if (x==0) {
   a = 2;
}
return (a+2);
```

mögliche uninitialisierte Benutzung

Dereferenzierung von NULL Pointern

```
int search(list_t *list, int x) {

while(list!=NULL) {
  list = list -> next;
  if (list -> elem == x) return 1;
}
return 0;
```

Dereferenzierung des NULL Pointers möglich

Andere Beispiele für Programmanalysen

- Cache Analyse
- Pipelineanalyse
- Sicherheitsanalysen

Statische PA <-> Dynamische PA

Dynamische PA
- z.B. profile basierte Programmoptimierung

  Aufsammeln von Informationen (z.B. Ausführungshäufigkeiten) anhand von einzelnen Beispielfällen

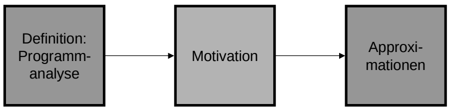
  Nachteil: Erzeugter Code muß trotzdem alle Fälle abdecken

Statische PA

  erzeugen der Information ohne das Programm auszuführen

  Optimierung gilt für alle möglichen Eingaben

Überblick



Beispiel

- Eingabe: Pascal ähnliches Programm mit Feldern
- Ausgabe: C ähnliches Programm mit Adreßarithmetik
- Entfernen von redundanten Berechnungen
- Vereinfachung der Adreßberechnung
- Folge von Analysen und Transformationen

Zeilenweise Ablage

a, b, c : **array** [0 : m, 0 : n] **of integer**

	a[0,0]	a[0,1]	...	a[0,n–1]	a[0,n]
i∗(n+1)	a[1,0]	a[1,1]	...	a[1,n–1]	a[1,n]
	⋮	j ⋮	a[i,j]	⋮	⋮
	a[m–1,0]	a[m–1,1]	...	a[m–1,n–1]	a[m–1,n]
	a[m,0]	a[m,1]	...	a[m,n–1]	a[m,n]

Elementweise Addition zweier Matrizen

```
i := 0;
while i <= m do
    ( j := 0;
      while j <= n do
        ( a[i, j] := b[i, j] + c[i, j];
          j := j + 1
        );
      i := i + 1
    );
```


Program converted to C-like Arrays

a[i, j] := b[i, j] + c[i, j];

```
i := 0;
while i <= m do
  ( j := 0 ;
    while j <= n do
      ( ta := Base(a) + i ∗ (n +1) + j;
        Cont(ta) := Cont(Base(b) + i ∗ (n + 1) + j)+
                    Cont(Base(c) + i ∗ (n + 1) + j);
        j := j + 1;
      );
    i := i + 1);
```


Analysis: Available Expressions

First computation

Recomputation

```
i := 0;
while i <= m do
  ( j := 0 ;
    while j <= n do
      ( ta := Base(a) + i ∗ (n +1) + j;
        Cont(ta):=Cont(Base(b) + i ∗ (n + 1) + j) +
                Cont(Base(c) + i ∗ (n + 1) + j);
        j := j + 1;
      );
    i := i + 1);
```

Common Subexpressions Eliminated

```
i := 0;
while i <= m do
   ( j := 0 ;
    while j <= n do
       ( t1 := i * (n +1) + j ;
         ta := Base(a) + t1 ;
         Cont(ta):=Cont(Base(b)+t1) + Cont(Base(c)+t1);
         j := j + 1;
       );
    i := i + 1);
```


Analysis: Loop Invariant Computation

```
i := 0;
while i <= m do
   ( j := 0 ;
    while j <= n do
       ( t1 := i * (n +1) + j ;
         ta := Base(a) + t1 ;
         Cont(ta):=Cont(Base(b)+t1) + Cont(Base(c)+t1);
         j := j + 1;
       );
    i := i + 1);
```


Invariant Code Motion

```
i := 0;
while i <= m do
   ( j := 0 ;
    t2 := i * (n + 1) ;
    while j <= n do
       ( t1 := t2 + j ;
         ta := Base(a) + t1 ;
         Cont(ta):=Cont(Base(b)+t1) + Cont(Base(c)+t1);
         j := j + 1;
       );
    i := i + 1);
```


Analysis: Induction Variable

Reaching Definitions

Loop Invariant expression

```
i := 0;
while i <= m do
   ( j := 0 ;
    t2 := i * (n + 1);
    while j <= n do
       ( t1 := t2 + j ;
         ta := Base(a) + t1 ;
         Cont(ta):=Cont(Base(b) + t1) + Cont(Base(c) + t1);
         j := j + 1;
       );
    i := i + 1);
```


Strength Reduction

```
i := 0; t3 := 0;
while i <= m do
   ( j := 0 ;
    t2 := t3;
    while j <= n do
       ( t1 := t2 + j ;
         ta := Base(a) + t1 ;
         Cont(ta):=Cont(Base(b) + t1) + Cont(Base(c) + t1);
         j := j + 1;
       );
    t3 := t3 + (n + 1);
    i := i + 1);
```


Copy Analysis

```
i := 0; t3 := 0;
while i <= m do
   ( j := 0 ;
    t2 := t3;
    while j <= n do
       ( t1 := t2 + j ;
         ta := Base(a) + t1 ;
         Cont(ta):=Cont(Base(b) + t1) + Cont(Base(c) + t1);
         j := j + 1;
       );
    t3 := t3 + (n + 1);
    i := i + 1);
```

Copy Propagation

```
i := 0; t3 := 0;
while i <= m do
   ( j := 0 ;
     t2 := t3;
     while j <= n do
        ( t1 := t3 + j  ;
          ta := Base(a) + t1 ;
          Cont(ta):=Cont(Base(b) + t1) + Cont(Base(c) + t1);
          j := j + 1;
        );
      t3 := t3 + (n + 1);
      i := i + 1);
```


Analysis: Dead Variable

```
i := 0; t3 := 0;
while i <= m do
   ( j := 0 ;
     t2 := t3;
     while j <= n do
        ( t1 := t3 + j  ;
          ta := Base(a) + t1 ;
          Cont(ta):=Cont(Base(b) + t1) + Cont(Base(c) + t1);
          j := j + 1;
        );
      t3 := t3 + (n + 1);
      i := i + 1);
```


Elimination of Useless Assignment

```
i := 0; t3 := 0;
while i <= m do
   ( j := 0 ;
     while j <= n do
        ( t1 := t3 + j  ;
          ta := Base(a) + t1 ;
          Cont(ta):=Cont(Base(b) + t1) + Cont(Base(c) + t1);
          j := j + 1;
        );
      t3 := t3 + (n + 1);
      i := i + 1);
```


Entfernen redundanter Berechnung

Transformation:

- einfügen einer temporären Variable
- vor der erste Berechnung des Ausdrucks wird die Variable gesetzt
- alle Benutzungen des Ausdrucks werden durch Benutzungen der Variable ersetzt

```
ta := Base(a) + i * (n +1) + j;
Cont(ta) := Cont(Base(b) +
               i * (n + 1) + j)+
               Cont(Base(c) +
               i * (n + 1) + j);
```

⇒

```
t1 := i * (n +1) + j;
ta := Base(a) + t1;
Cont(ta):=Cont(Base(b)+
          t1 +
          Cont(Base(c)+
          t1;
```


Entfernen redundanter Berechnung

Voraussetzung:

- Keine der im Ausdruck vorkommenden Varibalen ändert Ihren Wert zwischen erster und letzter Berechnung
- mindestens zwei Vorkommen

```
ta := Base(a) + i * (n +1) + j;
Cont(ta) := Cont(Base(b) +
               i * (n + 1) + j)+
               Cont(Base(c) +
               i * (n + 1) + j);
```

⇒

```
t1 := i * (n +1) + j;
ta := Base(a) + t1;
Cont(ta):=Cont(Base(b)+
          t1 +
          Cont(Base(c)+
          t1;
```


Herausziehen von Schleifeninvariantem Code

Transformation:

- Berechnung von Ausdrücken wird aus der Schleife herausgezogen

Voraussetzung:

- keine der im Ausdruck vorkommenden Variablen ändert ihren Wert innerhalb des Schleifenrumpfs

```
for (i=0;i<10;i++) {
  z = r*r;
  k = k*i;
}
```

⇒

```
z = r*r;
for (i=0;i<10;i++) {

   k = k*i;
}
```

Strength Reduction

Transformation:

- Vereinfachung von arithmetischen Berechnungen in Schleifen durch Ersetzen von monolitischen Berechnung im Schleifenrumpf durch iterative Berechnungen

```
i := 0;

while i <= m do
    t2 := i * (n + 1) ;
    ...
    i := i + 1;

end
```

⇒

```
i := 0;
t3 := 0;
while i <= m do
    t2 := t3 ;
    ...
    i := i + 1;
    t3 := t3 + (n + 1);
end
```


Strength Reduction

Voraussetzung:

- der zu vereinfachende Ausdruck ist eine Multiplikation oder ein Potenzierung
- der Ausdruck ist Funktion der Form $e = n*x + k$ oder $e = n^x$ in der Schleifendurchlaufzahl x
- n und k sind invariant im Schleifenrumpf

Entfernen von Redundanten Variablen

Transformation:

- Ersetze die Benutzungen einer Variable x durch Benutzungen der Variable y falls y eine Kopie von x ist

Voraussetzung:

- an allen Benutzungsstellen von y hat x denselben Wert wie x

```
t2 := t3;
...
t1 := t2 + j;
```

⇒

```
t2 := t3;
...
t1 := t3 + j;
```


Entfernen von Überflüssigen Zuweisungen

Transformation:

- Entferne Zuweisungen an Variablen, deren Wert nicht benutzt wird

Voraussetzung:

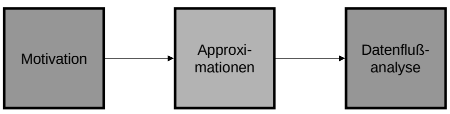
- es gibt keine Benutzung

```
x := 3;
a := z;
x := a;
```

⇒

```
a := z;
x := a;
```


Überblick

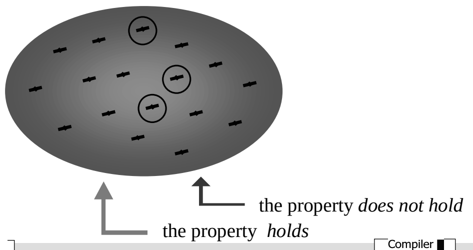


Das Halteproblem und seine Folgen...

- Alle interessanten Probleme sind nicht berechenbar.
- Daher sind approximative Lösungen nötig
- Irrtum ist nur auf der sicheren Seite erlaubt
- Sichere Seite hängt von der Verwendung der Information ab.
- Je weniger Irrtum, desto genauer die Analyse

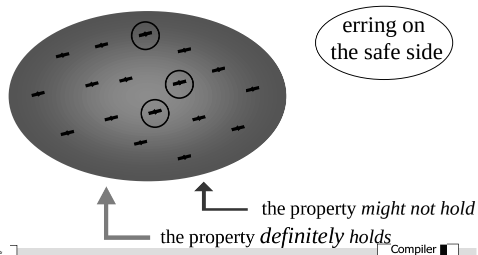
The nature of approximation

Exact answers

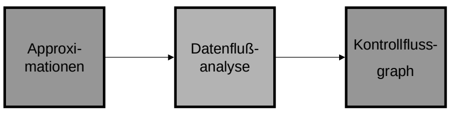


The nature of approximation

Approximate answers



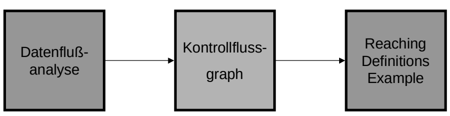
Überblick



Datenflußanalyse

- Eingeführt ca. 1973
- Ziel: Berechnung von Eigenschaften eines Programms für jeden Programmpunkt
- Arbeitet auf dem Kontrollflußgraph

Überblick



Kontrollflußgraph

Repräsentiert die Struktur eines Programms

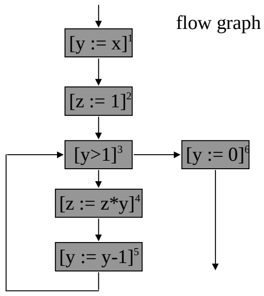
- Knoten repräsentieren Anweisungen oder Anweisungsteile
- Kanten repräsentieren mögliche Kontrollübergänge
- Kanten können Label tragen

Example

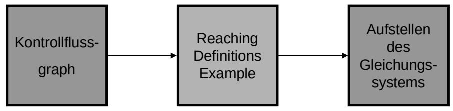
program
with labels

[y := x]^1;
[z := 1]^2;
<u>while</u> [y>1]^3
<u>do</u> ([z := z*y]^4;
[y := y-1]^5
);
[y := 0]^6

flow graph

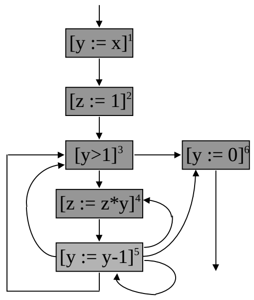


Überblick



Reaching Definitions

- Eine Zuweisung [x:=a]^l erreicht l' falls es eine Ausführungspfad gibt, auf dem x zuletzt in *l* zugewiesen wurde



Reaching Definitions

(x,?), (y,?), (z,?)

[y := x]^1;
[z := 1]^2;
<u>while</u> [y>1]^3
<u>do</u> ([z := z*y]^4;
[y := y-1]^5
);
[y := 0]^6

Reaching Definitions

(x,?), ~~(y,?)~~, (z,?)
(x,?), (y,1), (z,?)

[y := x]^1;
[z := 1]^2;
<u>while</u> [y>1]^3
<u>do</u> ([z := z*y]^4;
[y := y-1]^5
);
[y := 0]^6

Reaching Definitions

(x,?), (y,?), (z,?)
(x,?), (y,1), ~~(z,?)~~
(x,?), (y,1), (z,2)

[y := x]^1;
[z := 1]^2;
<u>while</u> [y>1]^3
<u>do</u> ([z := z*y]^4;
[y := y-1]^5
);
[y := 0]^6

Reaching Definitions

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

Reaching Definitions

(x,?), (y,?), (z,?)
[y := x]^1;
(x,?), (y,1), (z,?)
[z := 1]^2;
(x,?), (y,1), (z,2)
while [y>1]^3 (x,?), (y,1), ~~(z,2)~~
do ([z := z*y]^4; (x,?), (y,1), (z,4)
[y := y-1]^5
);
(x,?), (y,1), (z,2)
[y := 0]^6

Reaching Definitions

(x,?), (y,?), (z,?)
[y := x]^1;
(x,?), (y,1), (z,?)
[z := 1]^2;
(x,?), (y,1), (z,2)
while [y>1]^3 (x,?), (y,1), (z,2)
do ([z := z*y]^4; (x,?), ~~(y,1)~~, (z,4)
[y := y-1]^5 (x,?), (y,5), (z,4)
);
(x,?), (y,1), (z,2)
[y := 0]^6

Reaching Definitions

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

Reaching Definitions

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

Reaching Definitions

(x,?), (y,?), (z,?)
[y := x]^1;
(x,?), (y,1), (z,?)
[z := 1]^2;
(x,?), (y,1), (z,2), (y,5), (z,4)
while [y>1]^3 (x,?), (y,1), ~~(z,2)~~, (y,5), ~~(z,4)~~
do ([z := z*y]^4; (x,?), (y,1), (z,4), (y,5)
[y := y-1]^5 (x,?), (y,5), (z,4)
);
(x,?), (y,1), (z,2), (y,5), (z,4)
[y := 0]^6

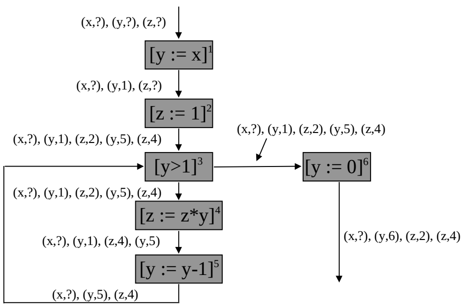
Reaching Definitions

Program	Reaching definitions
	(x,?), (y,?), (z,?)
[y := x]^1;	(x,?), (y,1), (z,?)
[z := 1]^2;	(x,?), (y,1), (z,2), (y,5), (z,4)
while [y>1]^3	(x,?), (y,1), (z,2), (y,5), (z,4)
do ([z := z*y]^4;	(x,?), ~~(y,1)~~, (z,4), ~~(y,5)~~
[y := y-1]^5	(x,?), (y,5), (z,4)
);	(x,?), (y,1), (z,2), (y,5), (z,4)
[y := 0]^6	

Reaching Definitions

Program	Reaching definitions
	(x,?), (y,?), (z,?)
[y := x]^1;	(x,?), (y,1), (z,?)
[z := 1]^2;	(x,?), (y,1), (z,2), (y,5), (z,4)
while [y>1]^3	(x,?), (y,1), (z,2), (y,5), (z,4)
do ([z := z*y]^4;	(x,?), (y,1), (z,4), (y,5)
[y := y-1]^5	(x,?), (y,5), (z,4)
);	(x,?), ~~(y,1)~~, (z,2), ~~(y,5)~~, (z,4)
[y := 0]^6	(x,?), (y,6), (z,2), (z,4)

Reaching Definitions



Reaching Definitions

Another *safe* solution - but *not* the best:

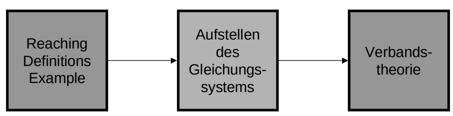
Program	Reaching definitions
	(x,?), (y,?), (z,?)
[y := x]^1;	(x,?), (y,1), (z,?)
[z := 1]^2;	(x,?), (y,1), (z,2), (y,5), (z,4)
while [y>1]^3	(x,?), (y,1), (z,2), (y,5), (z,4)
do ([z := z*y]^4;	(x,?), (y,1), (z,2), (y,5), (z,4)
[y := y-1]^5	(x,?), (y,1), (z,2), (y,5), (z,4)
);	(x,?), (y,1), (z,2), (y,5), (z,4)
[y := 0]^6	(x,?), (y,6), (z,2), (z,4)

Reaching Definitions

Unsafe solution:

Program	Reaching definitions
	(x,?), (y,?), (z,?)
[y := x]^1;	(x,?), (y,1), (z,?)
[z := 1]^2;	(x,?), (y,1), (z,2), (y,5), (z,4)
while [y>1]^3	(x,?), ~~(y,1)~~, ~~(z,2)~~, (y,5), (z,4)
do ([z := z*y]^4;	(x,?), ~~(y,1)~~, (z,4), (y,5)
[y := y-1]^5	(x,?), (y,5), (z,4)
);	(x,?), ~~(y,1)~~, ~~(z,2)~~, (y,5), (z,4)
[y := 0]^6	(x,?), (y,6), (z,2), (z,4)

Überblick



Berechnung der Datenflußinformation

- Extrahieren des Gleichungssystems
- Berechnen der kleinsten Lösung

$\downarrow$ $RD_{entry}(l)$
$[\ \dots\ ]^l$
$\downarrow$ $RD_{exit}(l)$

- remove killed variables
- add generated variables

$[y := x]^1$;
$[z := 1]^2$;
<u>while</u> $[y>1]^3$
<u>do</u> ($[z := z*y]^4$;
$[y := y-1]^5$
);
$[y := 0]^6$

$\downarrow$ $RD_{entry}(l)$
$[\ \dots\ ]^l$
$\downarrow$ $RD_{exit}(l)$

- remove killed variables
- add generated variables

$[y := x]^1$; ← $RD_{exit}(1) = (RD_{entry}(1) \setminus \{ (y,l) \mid l \in \mathbf{Lab} \}) \cup \{ (y,1) \}$
$[z := 1]^2$;
<u>while</u> $[y>1]^3$
<u>do</u> ($[z := z*y]^4$;
$[y := y-1]^5$
);
$[y := 0]^6$

$\downarrow$ $RD_{entry}(l)$
$[\ \dots\ ]^l$
$\downarrow$ $RD_{exit}(l)$

- remove killed variables
- add generated variables

$[y := x]^1$; ← $RD_{exit}(1) = (RD_{entry}(1) \setminus \{ (y,l) \mid l \in \mathbf{Lab} \}) \cup \{ (y,1) \}$
$[z := 1]^2$; ← $RD_{exit}(2) = (RD_{entry}(2) \setminus \{ (z,l) \mid l \in \mathbf{Lab} \}) \cup \{ (z,2) \}$
<u>while</u> $[y>1]^3$
<u>do</u> ($[z := z*y]^4$;
$[y := y-1]^5$
);
$[y := 0]^6$

$\downarrow$ $RD_{entry}(l)$
$[\ \dots\ ]^l$
$\downarrow$ $RD_{exit}(l)$

- remove killed variables
- add generated variables

$[y := x]^1$; ← $RD_{exit}(1) = (RD_{entry}(1) \setminus \{ (y,l) \mid l \in \mathbf{Lab} \}) \cup \{ (y,1) \}$
$[z := 1]^2$; ← $RD_{exit}(2) = (RD_{entry}(2) \setminus \{ (z,l) \mid l \in \mathbf{Lab} \}) \cup \{ (z,2) \}$
<u>while</u> $[y>1]^3$ ← $RD_{exit}(3) = RD_{entry}(3)$
<u>do</u> ($[z := z*y]^4$;
$[y := y-1]^5$
);
$[y := 0]^6$

$\downarrow$ $RD_{entry}(l)$
$[\ \dots\ ]^l$
$\downarrow$ $RD_{exit}(l)$

- remove killed variables
- add generated variables

$[y := x]^1$; ← $RD_{exit}(1) = (RD_{entry}(1) \setminus \{ (y,l) \mid l \in \mathbf{Lab} \}) \cup \{ (y,1) \}$
$[z := 1]^2$; ← $RD_{exit}(2) = (RD_{entry}(2) \setminus \{ (z,l) \mid l \in \mathbf{Lab} \}) \cup \{ (z,2) \}$
<u>while</u> $[y>1]^3$ ← $RD_{exit}(3) = RD_{entry}(3)$
<u>do</u> ($[z := z*y]^4$; ← $RD_{exit}(4) = (RD_{entry}(4) \setminus \{ (z,l) \mid l \in \mathbf{Lab} \}) \cup \{ (z,4) \}$
$[y := y-1]^5$ ← $RD_{exit}(5) = (RD_{entry}(5) \setminus \{ (y,l) \mid l \in \mathbf{Lab} \}) \cup \{ (y,5) \}$
);
$[y := 0]^6$ ← $RD_{exit}(6) = (RD_{entry}(6) \setminus \{ (y,l) \mid l \in \mathbf{Lab} \}) \cup \{ (y,6) \}$

$RD_{exit}(l')$ [...]$^{l'}$ [...]$^{l''}$ $RD_{exit}(l'')$

[...]^{l} $RD_{entry}(l)$

• join information from where control could come from

[y := x]^1;

[z := 1]^2;

<u>while</u> [y>1]^3

<u>do</u> ([z := z*y]^4;

[y := y-1]^5

);

[y := 0]^6

$RD_{exit}(l')$ [...]$^{l'}$ [...]$^{l''}$ $RD_{exit}(l'')$

[...]^{l} $RD_{entry}(l)$

• join information from where control could come from

[y := x]^1;

[z := 1]^2; ← $RD_{entry}(2) = RD_{exit}(1)$

<u>while</u> [y>1]^3

<u>do</u> ([z := z*y]^4;

[y := y-1]^5

);

[y := 0]^6

$RD_{exit}(l')$ [...]$^{l'}$ [...]$^{l''}$ $RD_{exit}(l'')$

[...]^{l} $RD_{entry}(l)$

• join information from where control could come from

[y := x]^1;

[z := 1]^2; ← $RD_{entry}(2) = RD_{exit}(1)$

<u>while</u> [y>1]^3 ← $RD_{entry}(3) = RD_{exit}(2) \cup RD_{exit}(5)$

<u>do</u> ([z := z*y]^4;

[y := y-1]^5

);

[y := 0]^6

$RD_{exit}(l')$ [...]$^{l'}$ [...]$^{l''}$ $RD_{exit}(l'')$

[...]^{l} $RD_{entry}(l)$

• join information from where control could come from

[y := x]^1;

[z := 1]^2; ← $RD_{entry}(2) = RD_{exit}(1)$

<u>while</u> [y>1]^3 ← $RD_{entry}(3) = RD_{exit}(2) \cup RD_{exit}(5)$

<u>do</u> ([z := z*y]^4; ← $RD_{entry}(4) = RD_{exit}(3)$

[y := y-1]^5 ← $RD_{entry}(5) = RD_{exit}(4)$

);

[y := 0]^6 ← $RD_{entry}(6) = RD_{exit}(5)$

[...]^{l} $RD_{entry}(l)$

• initially all variables are uninitialised

[y := x]^1; ← $RD_{entry}(1) = \{ (x,?), (y,?), (z,?) \}$

[z := 1]^2; ← $RD_{entry}(2) = RD_{exit}(1)$

<u>while</u> [y>1]^3 ← $RD_{entry}(3) = RD_{exit}(2) \cup RD_{exit}(5)$

<u>do</u> ([z := z*y]^4; ← $RD_{entry}(4) = RD_{exit}(3)$

[y := y-1]^5 ← $RD_{entry}(5) = RD_{exit}(4)$

);

[y := 0]^6 ← $RD_{entry}(6) = RD_{exit}(5)$

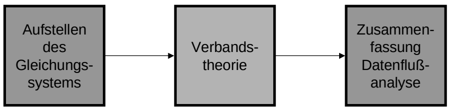
Reaching Definitions

Summary

- 12 sets:
  - $RD_{entry}(1), \ldots, RD_{exit}(6)$
- 12 equations:
  - $RD_j = F_j\,(RD_{entry}(1), \ldots, RD_{exit}(6))$
- one function:
  - $F: (\wp(Var_* \times Lab_*))^{12} \rightarrow (\wp(Var_* \times Lab_*))^{12}$
- we want the least fixed point of F

  --- but --- does it exist?

Überblick



Lattice theory

A complete lattice $(L, \sqsubseteq)$ is a set L with a *partial ordering* $\sqsubseteq$:

- reflexive: $l \sqsubseteq l$
- transitive: $l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \rightarrow l_1 \sqsubseteq l_3$
- anti-symmetric: $l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \rightarrow l_1 = l_2$

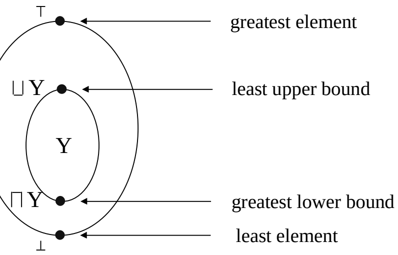
and such that all subsets Y have *least upper bounds* $\sqcup Y$:

- upper bound: $\forall l \in Y : l \sqsubseteq \sqcup Y$
- least: <u>if</u> $\forall l \in Y : l \sqsubseteq l_0$ <u>then</u> $\sqcup Y \sqsubseteq l_0$

Lattice theory

$(L, \sqsubseteq)$

complete lattice

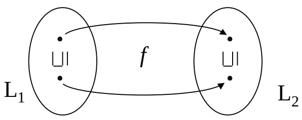


<u>Fact:</u> $((\wp(Var_* \times Lab_*))^{12}, \subseteq)$ is a complete lattice; the least element is $(\emptyset, \ldots, \emptyset)$

Lattice theory

A function f between two complete lattices $(L_1, \sqsubseteq_1)$ and $(L_2, \sqsubseteq_2)$ is a monotone function if

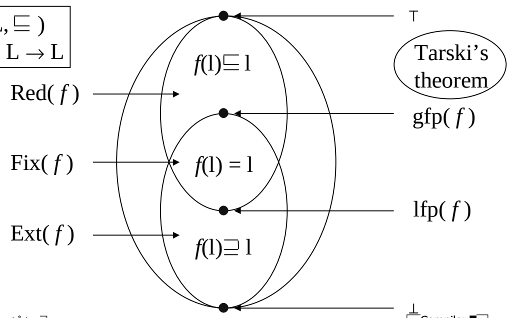
- $l_1 \sqsubseteq_1 l_2 \rightarrow f(l_1) \sqsubseteq_2 f(l_2)$



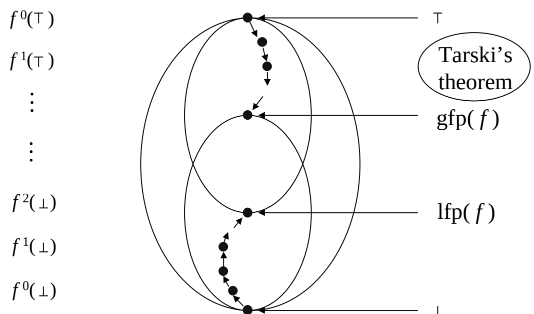
<u>Fact</u>: F is a monotone function

Lattice theory

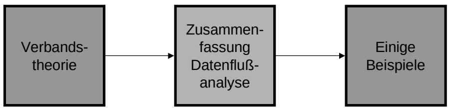
$(L, \sqsubseteq)$
$f : L \rightarrow L$



Lattice theory



Überblick



Datenflußanalyse

- Vollständiger Verband mit aufsteigender Kettenbedingung
- Gleichungssystem
  - Transferfunktionen
  - combine Funktion
- Kontrollflußgraph

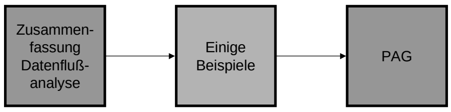
⇨ Kleinster Fixpunkt existiert und ist endlich berechenbar

Datenflußanalyse

Zusätzlich:

- Richtung (vorwärts, rückwärts)
- kleinster/größter Fixpunkt
  - Initialisierung
- Wert für den Startknoten

Überblick



Reaching Definitions (RD)

- Def: Eine Zuweisung $[x:=a]^l$ erreicht l', falls es einen Ausführungspfad gibt, auf dem x zuletzt in *l* zugewiesen wurde
- Verband: $\wp(\mathrm{Var}_* \times \mathrm{Lab}_*)$
- Fixpunkt: kleinster ⇨ init = bot
- Richtung: Vorwärtsproblem
- combine: lub = Vereinigung
- Transferfunktion: $[\, y:=\ldots\,]^l$

⇨ $RD_{exit}(l) = (RD_{entry}(l) \setminus \{ (y,l) \mid l \in \mathbf{Lab} \}) \cup \{ (y,l) \}$

Available Expressions (AE)

- Def: Ein Ausdruck *e* ist an einem Programmpunkt *l* verfügbar, falls auf allen Ausführungspfaden zu *l* *e* berechnet wurde und sich nicht verändert hat.
- Verband: $\wp(\mathrm{Aexp}_*)$
- Fixpunkt: größter ⇨ init = top
- Richtung: Vorwärtsproblem
- combine: glb = Schnitt
- Transferfunktion: $[\, y:= a\,]^l$

⇨ $AE_{exit}(l) = (AE_{entry}(l) \setminus \{ a' \in \mathrm{Aexp}_* \mid y \in \mathrm{Var}(a') \}) \cup \{ a' \in \mathrm{subexp}(x) \mid y \notin \mathrm{Var}(a') \}$

Dagstuhl LPM –21-Nov-00 85

Very Busy Expressions (VB)

- Def: Ein Ausdruck *e* ist an einem Programmpunkt *l* very busy, falls auf allen Ausführungspfaden von *l* zum Ende *e* benutzt wird bevor er sich verändert
- Verband: $\wp(Aexp_*)$
- Fixpunkt: größter ➪ init = top
- Richtung: Rückwärtsproblem
- combine: glb = Schnitt
- Transferfunktion: $[\ y{:=}\ a\ ]^l$

➪ $VB_{entry}(l) = (VB_{exit}(l) \setminus \{ a' \in Aexp_* \mid y \in Var(a') \}) \cup \{ subexp(x) \}$

Dagstuhl LPM –21-Nov-00 86

Live Variable Analysis (LV)

- Def: Eine Variable *v* ist an einem Programmpunkt *l* lebendig, falls es eine Ausführungspfaden von *l* zu einer Benutzung von *v* ohne vorherige Setzung gibt
- Verband: $\wp(Var_*)$
- Fixpunkt: kleinster ➪ init = bot
- Richtung: Rückwärtsproblem
- combine: lub = Vereinigung
- Transferfunktion: $[\ y{:=}\ a\ ]^l$

➪ $LV_{entry}(l) = (LV_{exit}(l) \setminus \{ y \}) \cup \{ Vars(a) \}$

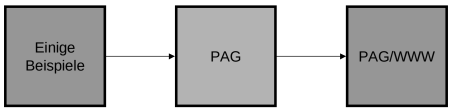
Dagstuhl LPM –21-Nov-00 87

Klassische Analysen

	Vereinigung	Schnitt
Vorwärts	RD	AE
Rückwärts	LV	VB

Dagstuhl LPM –21-Nov-00 88

Überblick



Dagstuhl LPM –21-Nov-00 89

Was ist PAG

- Generiert Implementierungen von monotonen Datenflußproblemen
- Nimmt eine Spezifikation als Eingabe
- Erzeugt ANSI C Code, der in einen Compiler integriert werden kann
- Verwendbar für Lehre, Forschung und für industrielle Produkte
- Unterstützt imperative, objektorientierte und logische Programmiersprachen

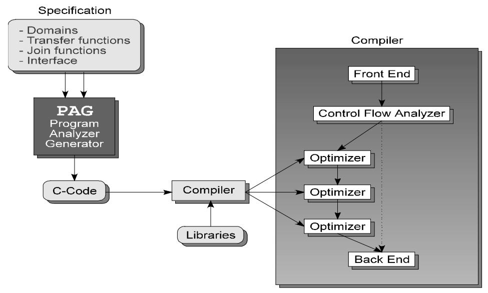
Dagstuhl LPM –21-Nov-00 90

Implementation Problems without PAG

Hand implementation of data flow analyses are

- complicated
- error prone
- difficult to debug
- difficult to maintain
- large similar parts had to be implemented again and again

Overview



PAG offers

- specification of the set of abstract values through inductive definitions
- functional language for definition of transfer functions
- set of parameters
- different interprocedural solution algorithms
- simple interface to the control flow graph
- graphical debugging
- simplifies maintenance

Advantages of using PAG

- Generated analyzers can handle large programs.
- Reduction of implementation times of complex analyses due to the power of the specification mechanism.
- Newly developed analyses can be tested easily on a large set of real world programs.
- Development of provably correct optimisation phases for compilers is possible now due to the underlying theory of abstract interpretation.

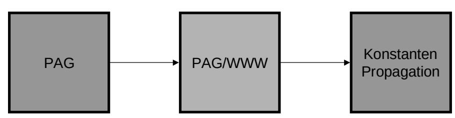
Generated Analyzers

- conditional constant propagation
- shape analysis
- memory error detection
- cache behaviour analysis
- pipeline behaviour analysis
- strongly live variables
- interval analysis
- classical bitvector analyses

PAG Users

- ESPRIT Project COMPARE
- DFG Transfer Project #14
- ESPRIT Project JOSES
- 16 site licenses
- In the PAG web frontend PAG/WWW http://pag.cs.uni-sb.de
- several uses in tutorials on program analysis
- AbsInt GmbH

Überblick



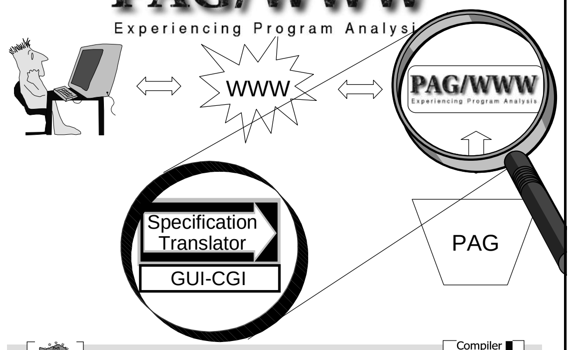
PAG/WWW

Experiencing Program Analysis

- WWW Interface to PAG
- build on top of the full PAG system
- restricted features
- simplified specification language
- fixed input language
- designed for teaching and understanding program analysis

PAG/WWW

Experiencing Program Analysis



PAG

- full system not bound to a specific language
- additional specification features
- more complicated syntax
- intended to work in a compiler
- also for research purposes
- different interprocedural iteration algorithms

PAG/WWW

Experiencing Program Analysis

- for WHILE programs only
- restricted specification languages
- restricted syntax
- additionally predefined functionality
- intended for educational purposes
- restricted to a certain iteration algorithm

PAG Web Frontend

www.cs.uni-sb.de/~martin/pag

The Web Frontend offers

- Selection of an analysis from a set of predefined analyses
- Selection of a test program for the chosen analysis
- View of analysis results
- Visualisation of the analysis steps
- Possibility to define and edit new analyses and test programs
- Good for educational purpose

The PAG Web Frontend gives a deep insight in how to design data flow analyses with the instruments supplied by the PAG-Generator Technology.

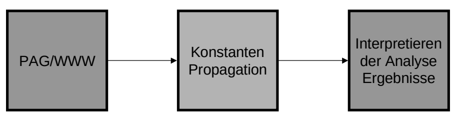
PAG/WWW Reaching Definitions

PAG/WWW

Experiencing Program Analysis

Demonstration der Reaching Definitions

Überblick

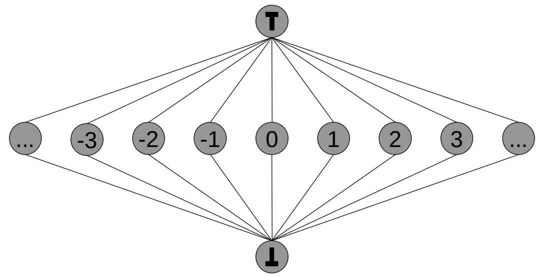


Konstantenpropagation (CP)

- Ziel: Berechne für jeden Programmpunkt Abbildung von Variablen auf konstante Werte (oder top falls nicht konstant)
- Verband: $Var_* \to \mathbb{N}^{\top}_{\bot}$
- Fixpunkt: kleinster ➪ init = bot
- Richtung: Vorwärtsproblem
- combine: lub
- Transferfunktion: $[\ y := a\ ]^l$

➪ $CP_{exit}(l) = CP_{entry}(l)\ [\ y \to [|\ a\ |]\ ]$

Flat



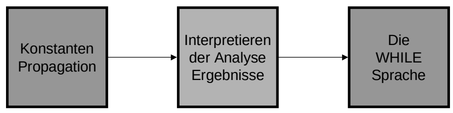
Konstantenpropagation

PAG/WWW
Experiencing Program Analysis

Demonstration der Konstantenpropagation

Textuelle Analyseergebnisse

Überblick



Interpreting textual Analysis Results

- automatically labeled program
- showing Entry (*Analysis*$_\circ$) and Exit (*Analysis*$_\bullet$) information
- procedure parameters are underlined
- [BOT:x->7] is a function mapping x to 7 and everything else bot (see constant propagation)

Konstantenpropagation

PAG/WWW
Experiencing Program Analysis

Demonstration der Konstantenpropagation

Graphische Analyseergebnisse

Interpreting graphical Analysis Results

- a picture for each computation step
- the Exit information is shown just beside the outgoing edges
- the Entry information is not displayed
- colors of nodes and information:
  - red for the information that is about to be changed
  - blue for nodes contained in the worklist
- nodes labeled with elementary statements and labels
- D $\rightarrow \perp$ is equivalent to [->BOT:x->7]
  x $\rightarrow$ 7

Überblick

Interpretieren der Analyse Ergebnisse → Die WHILE Sprache → Spezifikation der Analysen

While Language

Pascal like syntax but

- no variable declarations
- no nested procedures
- procedures have exactly one parameter
- no result parameter
- parentheses around blocks
- program result returned in the 'result' variable

While Statements

- **skip**
- <id> **:=** <aexp>
- **if** <bexp> **then** <block>
- **if** <bexp> **then** <block> **else** <block>
- **while** <bexp> **then** <block>
- **call** <id> (<aexp>)

- <block> $\rightarrow$ (<stmt>$^+$) **|** <stmt>

Editing While Programs/Specifications

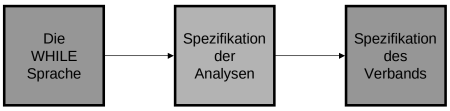
```
/* 10 */ skip
/* 11 */ skip

/* 10 */ skip
x := 3
/* 11 */ skip
```


WHILE

PAG/WWW
Experiencing Program Analysis

Demonstration der WHILE Sprache

Überblick



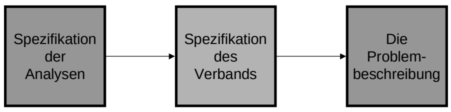
Specifying Analyses

- specialized functional language FULA
- allows concise specifications
- direct support of distributive analysis frameworks
- ML like syntax
- easy composition of lattices
- efficient implementation

Parts of Specification

- TYPE: define the analysis lattice
- PROBLEM: define analysis parameters
- TRANSFER: define the transfer functions
- SUPPORT: define additional functions

Überblick



Predefined Datatypes

snum	signed int
bool	boolen
str	strings
Label	program lables ($Lab_\star$)
Var	program variables ($Var_\star$)
Proc	program procedures
Expression	Non trivial program expressions ($Aexp_\star$)

Lattice Construction

set(<Id>)	Set over <Id>
list(<Id>)	List over <Id> NOT a lattice!
<Id$_1$> * <Id$_2$> ...	Tupel space
<Id$_1$> -> <Id$_2$>	Function space
flat(<Id>)	Flat lattice
lift(<Id>)	Lifted lattice

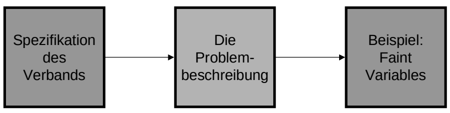
Predefined Sets

LabelSet	= set(Label)
VarSet	= set(Var)
ProcSet	= set(Proc)
ExpressionSet	= set(Expression)
ExpressionList	= list(Expression)

Example: Live Variables

```
TYPE

 VarSetLifted = lift(VarSet)
```


Überblick

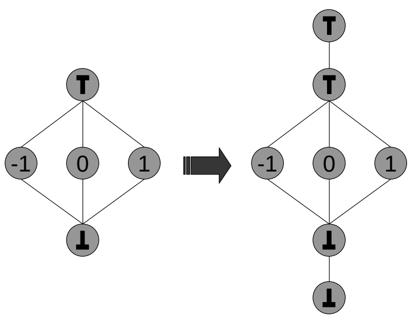


Excursion: Solution Algorithm

- in PAG the worklist is initialized only with the start node
- the programmer has to make sure that each node of interest is visited at least once

$\Rightarrow$ add extra $\perp$ if the $\perp$ of the lattice has a meaningful value

Lift



Problem Section

direction	forward/backward
carrier	analyis lattice
init	initial value
init_start	value for the extremal node
combine	combination function

Example: Live Variables

```
PROBLEM Live_Variables

 direction  : backward
 carrier    : VarSetLifted
 init       : bot
 init_start : lift({})
 combine    : lub
```


FULA Basics

- primitives: snum, str, bool
- sets: { <e_1>, ..., <e_n> } {}
- lattice elements **top bot**
- lists: [<e_1>, ..., <e_n>] [] <e>:list
- tuples: (<e_1>, ..., <e_n>)
- functions: static/dynamic
  f(<e>) [-> <e>] f\[<e_1>-><e_2>]

FULA Control Structures

- if ... then ... [else ...] endif
- case <e_1>, ..., <e_n> of
  <p_1>,..., <p_n> => <e> ...
  endcase
- let <p_1> = <e_1>, ... , <p_n> = <e_n> in <e>
- let < p_1> <= <e_1>, ... in <e> ⇔
  if <e_1> = top then top else
  if <e_1> = bot then bot else
  let < p_1> = drop(<e_1>)
  in <e> endif endif

TRANSFER Section

- define the exit value in terms of the entry value (@)
  (or vice versa for backward problems)
- definition by cases for the WHILE statements
- optional matching of the edge type

Example: Live Variables

```
TRANSFER
IF(exp) =
 if @ = top then top
 else if @ = bot then bot
 else let lifeVars = drop(@) in
    lift(lifeVars lub
        variables(exp))
 endif endif
```


Example: Live Variables

```
let lifeVars <= @ in
```

```
TRANSFER
IF(exp) =
 if @ = top then top
 else if @ = bot then bot
 else let lifeVars = drop(@) in
    lift(lifeVars lub
        variables(exp))
 endif endif
```

Example: Live Variables

```
TRANSFER
IF(exp) =

 let lifeVars <= @ in
    lift(lifeVars lub
        variables(exp))
```


Example: Live Variables

```
ASSIGN(var, exp) =
let lifeVars <= @  in
 lift((lifeVars - var) lub
         variables(exp))
```


FULA Operators

- =, !=
- <, <=, >, >= for lattice and snum
- boolean &&, ||, !
- Tupel selection (x,y)#1
- lub, glb for lattices
- member test x?{x,y}
- lift(x) drop(x)
- arithmetic operators
- + for string concatenation, set union, set insertion, list append

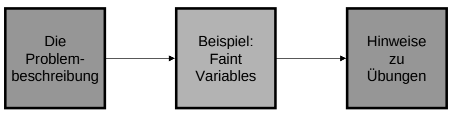
SUPPORT Section

- define additional functions
- each function needs a type declaration
- definition by cases possible

Example: Live Variables

```
variables :: Expression -> VarSet
variables(expression) =
  { expVar(exp) | exp in
    subExpressions(expression);
    expType(exp) = "VAR" }
```


Überblick



Faint Variables

Extension of the Live Variable problem

```
z := x;
y := z;
```

x is considered to be alive at the beginning

⇒ consider the variables used in an assignment to be alive only if the variable on the left side is alive

Changing the extremal Value

- assume that in the end of the program the variable **`result`** is alive
- the variable is accessible directly in FULA through **`result`**

Faint Variables: Solution

```
init_start : lift({result})

ASSIGN(variable, exp) =
let lifeVars <= @  in
 if variable ? lifeVars then
  lift((lifeVars - var) lub
         variables(exp))
 else @
 endif
```


PAG/WWW

PAG/WWW
Experiencing Program Analysis

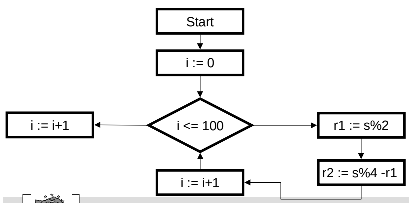
Demonstration der Faint Variables

Überblick

Beispiel: Faint Variables → Hinweise zu Übungen → Zusammen-fassung

Aufgabe 1

- Dominatoranalyse: Ein Knoten y wird von einem Knoten x dominiert, wenn x auf allen Pfaden vom Startknoten zu y liegt



Hinweise: Aufgabe 1

- Benutzen Sie die vordefinierte Variable **`label`**, die das aktuelle Label enthält

Aufgabe 2

- Bedingte Kontantenpropagation: Verbessern Sie die Konstantenpropagation

X:=3		X → 3
If x=3 then	X → 3	X → 3
X:=4 else	X → 3	X → 4
x:=5	X → 3	X → 5
y := x	X → TOP	X → TOP
		X → TOP

Hinweise: Aufgabe 2

- Benutzen Sie eine zusätzliche lift Operation um ein Element ⊥ einzuführen, dass die “nicht Erreichbarkeit” symbolisiert
- Modifizieren Sie die Transferfunktionen entsprechen (drop,lift)
- Schreiben Sie zwei Transferfunktionen für das IF

```
IF(exp),true_edge
IF(exp),false_edge
```


Aufgabe 3

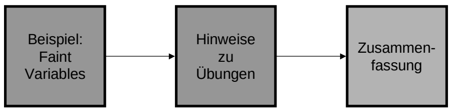
- Vorzeichenanalyse: Bestimme die Menge der Vorzeichen, die eine Variable annehmen kann.
- Beispiel: {-2,-1,1} -> {-,+}

Hinweise: Aufgabe 3

- Benutzen Sie die Konstanten-propagation als Grundlage
- Repräsentieren sie die Vorzeichen -,+,0 als Zahlen -1,0,1
- Verband: Abbildung von Variablen nach Mengen von Vorzeichen

```
vz_set = set(snum)
constfunc = Var -> vz_set
```


Überblick



Zusammenfassung

- Anwendungen der Programmanalyse
- Techniken zur Implementierung von Programmanalysen
- Werkzeug zur einfachen Erzeugung von Programmanalysen