Intra-procedural Data Flow Analysis

Introduction

Hanne Riis Nielson and Flemming Nielson

email: `{riis,nielson}@imm.dtu.dk`

Informatics and Mathematical Modelling
Technical University of Denmark

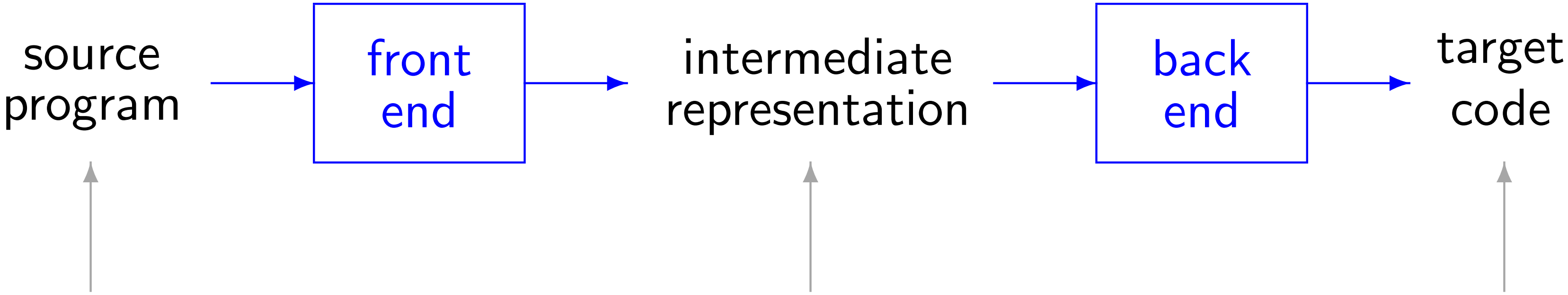
Copyright

This set of transparencies is mainly based on

> Flemming Nielson, Hanne Riis Nielson, Chris Hankin:
> Principles of Program Analysis.
> Springer, 1999; ISBN 3-540-65410-0.

The handout version of the transparancies is available from the webpage `http://www.imm.dtu.dk/˜riis/ppa.htm`.

The Setting: Optimising Compilers


source program

front end

intermediate representation

back end

target code

the programmer can:

- profile program
- change algorithm
- transform program

beyond the compiler

the compiler can:

- improve loops
- procedure calls
- address calculations

architecture independent

the compiler can:

- use registers
- instruction selection
- peephole optimisation

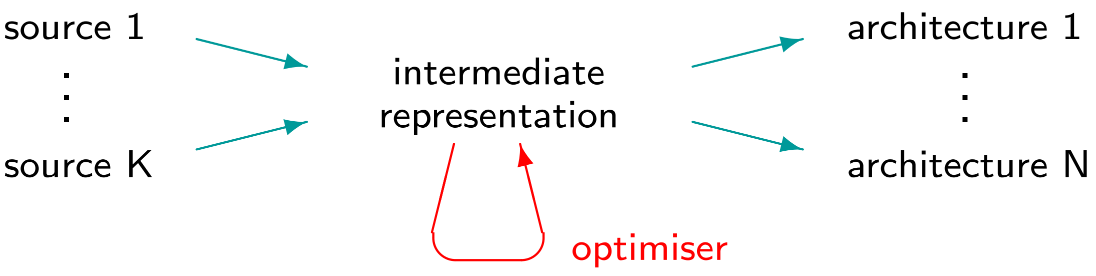
architecture dependent

Criteria for Optimisations

- an optimisation must preserve the meaning of programs
  - must not change the output for any input
  - must not cause errors (e.g. division by 0) that were not present before
- an optimisation must speed up programs by a measurable amount
  - occationally one optimise for space
  - not every optimisation succeeds in improving every program
  - occationally an “optimisation” may slow down a program slightly but this is acceptable as long as it improves the avarage speed up
- an optimisation must be worth the effort
  - the additional time spend by the compiler must be repayed when running the target program

The Level of Optimisations

- source level: language dependent
  - e.g. references to array elements give rise to redundant computations
    - ∗ Pascal and Fortran programs cannot be optimised at the source levels
    - ∗ C programs allow both kinds of references and can be optimised
- intermediate level: language and architecture in dependent



- target/low level: architecture dependent
  - less likely to be portable

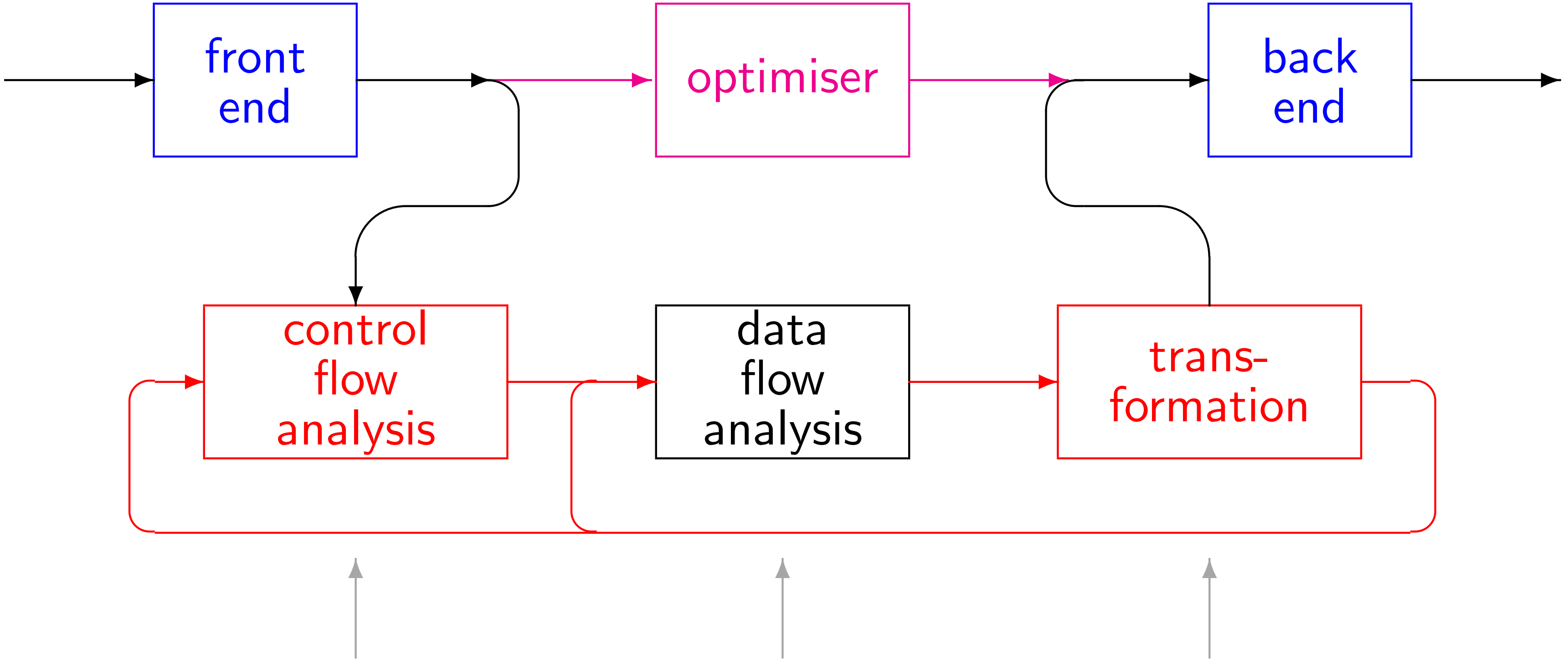
What Optimisations are Worthwhile?

Folklore: most programs spend 90% of their execution time in 10% of the code.

- Inner loops are good candidates
- Experiment, profiling, collect statistics, . . .
- Build on other people's experience, see e.g.

  Steven S. Muchnick:
  Advanced Compiler Design and Implementation
  Morgan Kaufmann, 1997

Organisation of an Optimiser



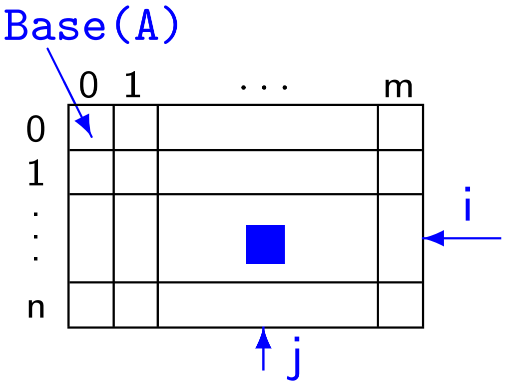
aim: to discover the hierarchical flow of control

aim: to determine information about data

aim: to modify the program

Example

(ack. Reinhard Wilhelm)



Algol-like arrays:

```
i := 0;
while i <= n do
   j := 0;
   while j <= m do
      A[i,j] := B[i,j] + C[i,j];
      j := j+1
   od;
   i := i+1
od
```

C-like arrays:

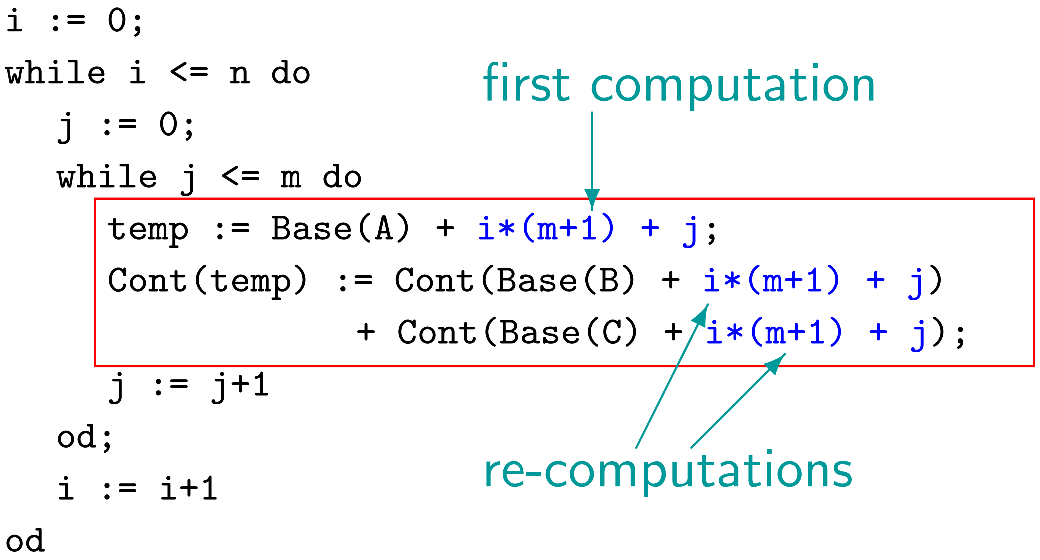
```
i := 0;
while i <= n do
   j := 0;
   while j <= m do
      temp := Base(A) + i * (m+1) + j;
      Cont(temp) := Cont(Base(B) + i * (m+1) + j)
                  + Cont(Base(C) + i * (m+1) + j);
      j := j+1
   od;
   i := i+1
od
```

Typical Optimisations

- Avoid redundant computations
  - reuse available results
  - move loop invariant computations out of loops
- Avoid superfluous computations
  - results known not to be needed
  - results known already at compile time

```
i := 0;
while i <= n do
   j := 0;
   while j <= m do
      temp := Base(A) + i * (m+1) + j;
      Cont(temp) := Cont(Base(B)
                             + i * (m+1) + j)
                       + Cont(Base(C)
                             + i * (m+1) + j);
      j := j+1
   od;
   i := i+1
od
```

Available Expressions Analysis



```
i := 0;
while i <= n do
  j := 0;
  while j <= m do
     temp := Base(A) + i*(m+1) + j;
     Cont(temp) := Cont(Base(B) + i*(m+1) + j)
                 + Cont(Base(C) + i*(m+1) + j);
     j := j+1
  od;
  i := i+1
od
```

Common subexpression elimination:

```
t1 := i * (m+1) + j;
temp := Base(A) + t1;
Cont(temp) := Cont(Base(B)+t1)
           + Cont(Base(C)+t1);
```

Detection of Loop Invariants

```
i := 0;
while i <= n do
   j := 0;
   while j <= m do
      t1 := i * (m+1) + j;
      temp := Base(A) + t1;
      Cont(temp) := Cont(Base(B) + t1)
                  + Cont(Base(C) + t1);
      j := j+1
   od;
   i := i+1
od
```

loop invariant

Invariant code motion:

```
t2 := i * (m+1);
while j <= m do
   t1 := t2 + j;
   temp := ...
   Cont(temp) := ...
   j := ...
od
```

Detection of Induction Variables

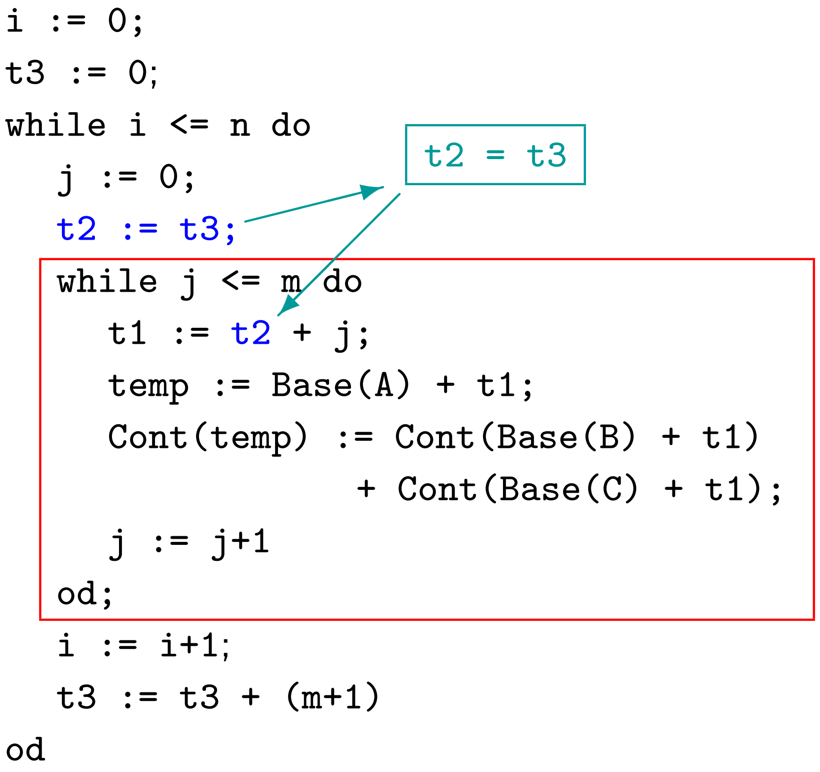
```
i := 0;
while i <= n do
   j := 0;
   t2 := i * (m+1);
   while j <= m do
      t1 := t2 + j;
      temp := Base(A) + t1;
      Cont(temp) := Cont(Base(B) + t1)
                    + Cont(Base(C) + t1);
      j := j+1
   od;
   i := i+1
od
```

induction variable

Strength reduction:

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

Copy Analysis



```
i := 0;
t3 := 0;
while i <= n do
   j := 0;
   t2 := t3;
   while j <= m do
      t1 := t2 + j;
      temp := Base(A) + t1;
      Cont(temp) := Cont(Base(B) + t1)
                  + Cont(Base(C) + t1);
      j := j+1
   od;
   i := i+1;
   t3 := t3 + (m+1)
od
```

Copy propagation:

```
while j <= m do
   t1 := t3 + j;
   temp := ...;
   Cont(temp) := ...;
   j := ...
od
```

Live Variables Analysis

```
i := 0;
t3 := 0;
while i <= n do
   j := 0;
   t2 := t3;
   while j <= m do
      t1 := t3 + j;
      temp := Base(A) + t1;
      Cont(temp) := Cont(Base(B) + t1)
                        + Cont(Base(C) + t1);
      j := j+1
   od;
   i := i+1;
   t3 := t3 + (m+1)
od
```

dead variable

Dead code elimination:

```
i := 0;
t3 := 0;
while i <= n do
   j := 0;
   while j <= m do
      t1 := t3 + j;
      temp := Base(A) + t1;
      Cont(temp) := Cont(Base(B) + t1)
                        + Cont(Base(C) + t1);
      j := j+1
   od;
   i := i+1;
   t3 := t3 + (m+1)
od
```

Analyses and Transformations

Data Flow Analysis	Transformation
Available expressions analysis	Common subexpression elimination
Detection of loop invariants	Invariant code motion
Detection of induction variables	Strength reduction
Copy analysis	Copy propagation
Live variables analysis	Dead code elimination
...	...

The Essence of Program Analysis

Program analysis offers techniques for predicting
statically at compile-time
safe & efficient approximations
to the set of configurations or behaviours arising
dynamically at run-time

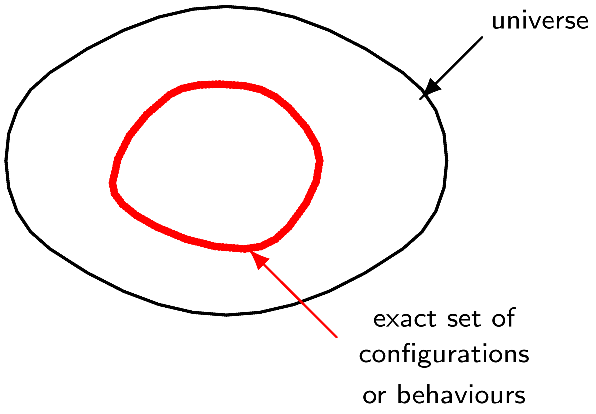
we cannot expect exact answers!

Safe: faithful to the semantics

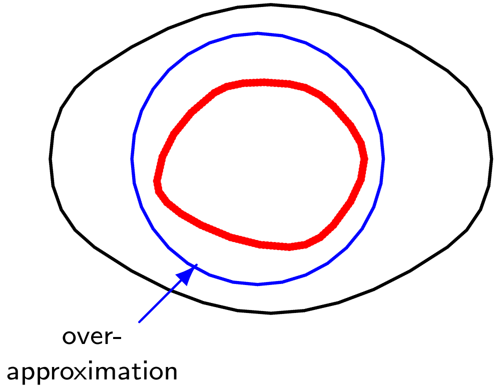
Efficient: implementation with
- good time performance and
- low space consumption

The Nature of Approximation

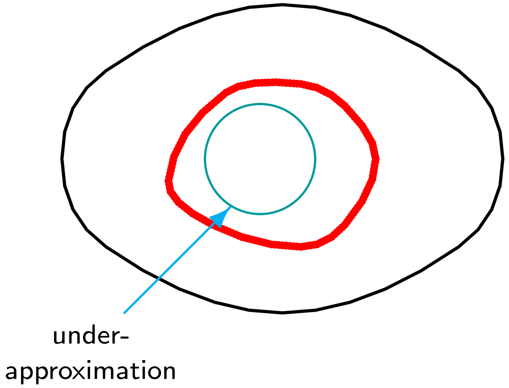
The exact world

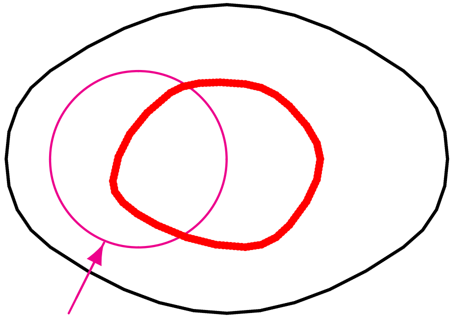


Over-approximation



Under-approximation





unacceptable!

Slogans: Err on the safe side!

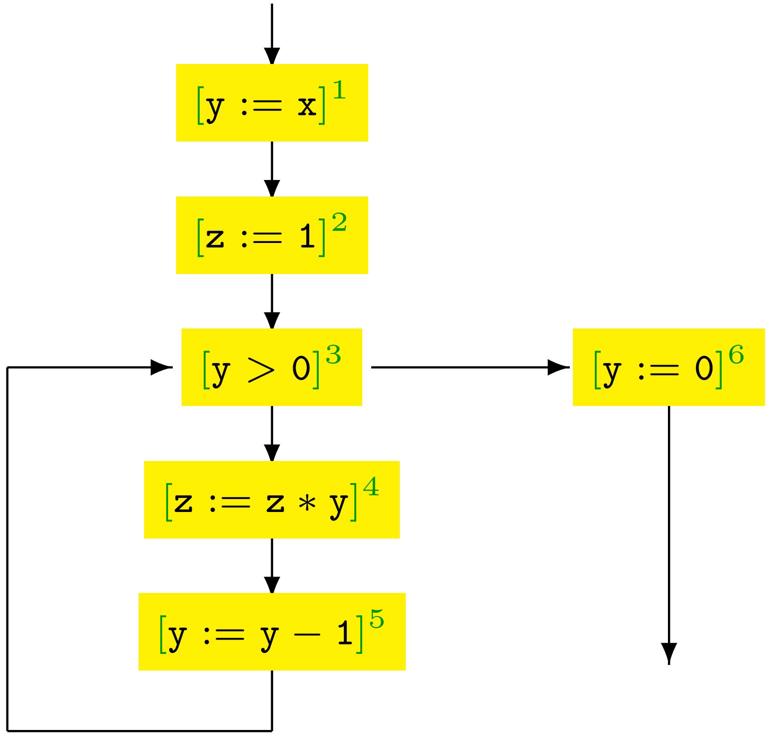
Trade precision for efficiency!

Example

Program with labels for elementary blocks:

```
$[y := x]^1$;
$[z := 1]^2$;
while $[y > 0]^3$ do
    $[z := z * y]^4$;
    $[y := y - 1]^5$
od;
$[y := 0]^6$
```

Flow graph:

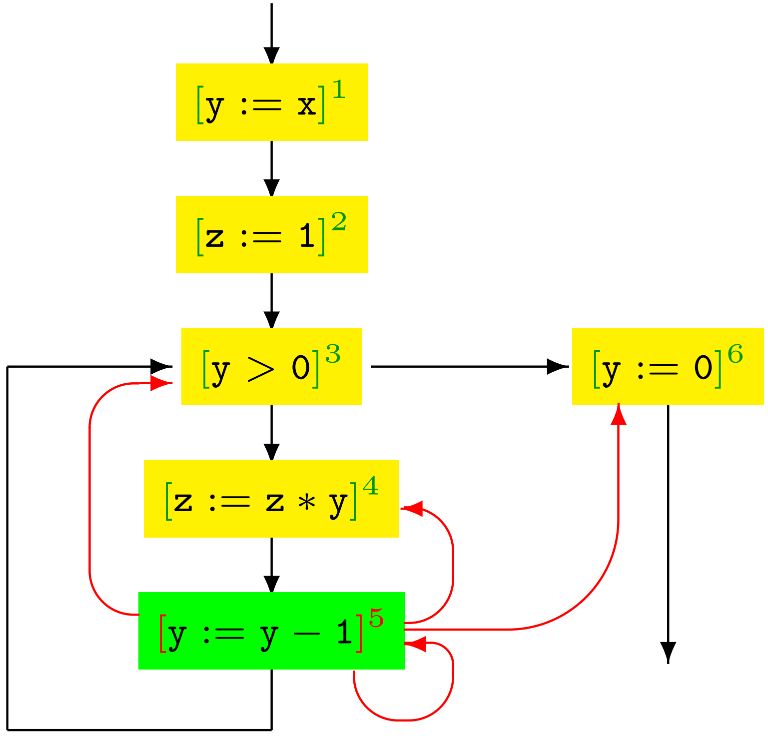


Example: Reaching Definitions Analysis

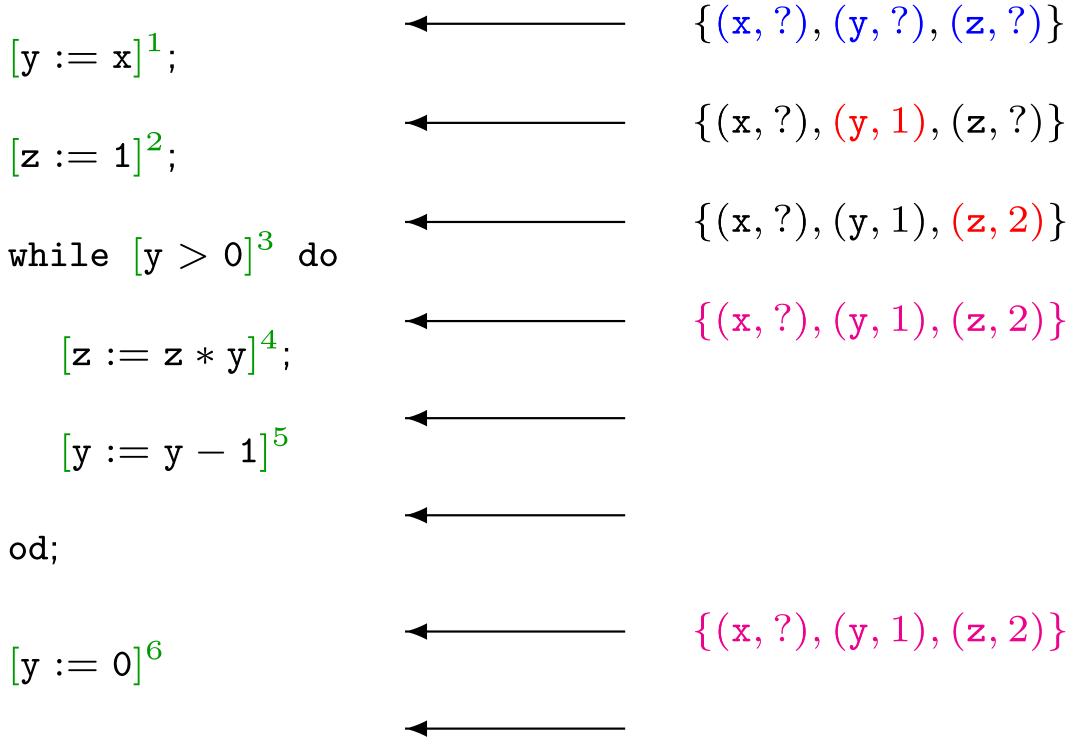
Problem: which definitions reach which program points

For a simple while language:
a definition of a variable x is an assignment $[x := a]^{\ell}$ to x

The assignment $[x := a]^{\ell}$ reaches ℓ' if there is an execution where x was last assigned at ℓ



Analysing the Program by Hand (1)



← $\{(x, ?), (y, ?), (z, ?)\}$

$[y := x]^1;$

← $\{(x, ?), (y, 1), (z, ?)\}$

$[z := 1]^2;$

← $\{(x, ?), (y, 1), (z, 2)\}$

while $[y > 0]^3$ do

← $\{(x, ?), (y, 1), (z, 2)\}$

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

←

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

←

od;

← $\{(x, ?), (y, 1), (z, 2)\}$

$[y := 0]^6$

←

Analysing the Program by Hand (2)

$\{(x, ?), (y, ?), (z, ?)\}$

$[\mathtt{y} := \mathtt{x}]^1$;

$\{(x, ?), (y, 1), (z, ?)\}$

$[\mathtt{z} := \mathtt{1}]^2$;

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

while $[\mathtt{y} > \mathtt{0}]^3$ do

$\{(x, ?), (y, 1), (z, 2)\}$

$[\mathtt{z} := \mathtt{z} * \mathtt{y}]^4$;

$\{(x, ?), (y, 1), (z, 4)\}$

$[\mathtt{y} := \mathtt{y} - \mathtt{1}]^5$

$\{(x, ?), (y, 5), (z, 4)\}$

od;

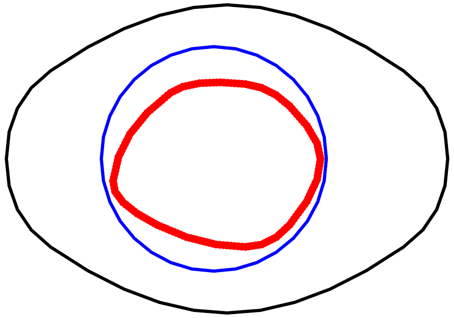
$\{(x, ?), (y, 1), (z, 2)\}$

$[\mathtt{y} := \mathtt{0}]^6$

Analysing the Program by Hand (3)

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

The Best Solution



$\longleftarrow \quad \{(x,?),(y,?),(z,?)\}$

$[\mathtt{y := x}]^1;$

$\longleftarrow \quad \{(x,?),(y,1),(z,?)\}$

$[\mathtt{z := 1}]^2;$

$\longleftarrow \quad \{(x,?),(y,1),(z,2),(y,5),(z,4)\}$

$\mathtt{while}\ [\mathtt{y > 0}]^3\ \mathtt{do}$

$\longleftarrow \quad \{(x,?),(y,1),(z,2),(y,5),(z,4)\}$

$\quad [\mathtt{z := z * y}]^4;$

$\longleftarrow \quad \{(x,?),(y,1),(y,5),(z,4)\}$

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

$\longleftarrow \quad \{(x,?),(y,5),(z,4)\}$

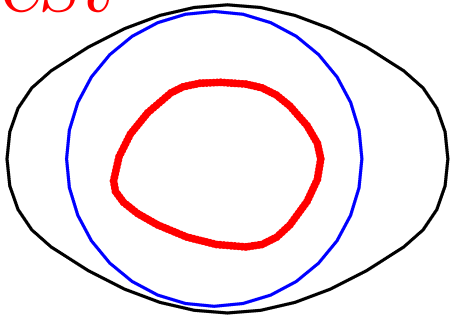
od;

$\longleftarrow \quad \{(x,?),(y,1),(z,2),(y,5),(z,4)\}$

$[\mathtt{y := 0}]^6$

$\longleftarrow \quad \{(x,?),(y,6),(z,2),(z,4)\}$

A Safe Solution — but not the Best



← $\{(x,?), (y,?), (z,?)\}$

$[y := x]^1$;

← $\{(x,?), (y,1), (z,?)\}$

$[z := 1]^2$;

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

while $[y > 0]^3$ do

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

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

← $\{(x,?), (y,1), (y,5), \boxed{(z,2)}, (z,4)\}$

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

← $\{(x,?), \boxed{(y,1)}, (y,5), \boxed{(z,2)}, (z,4)\}$

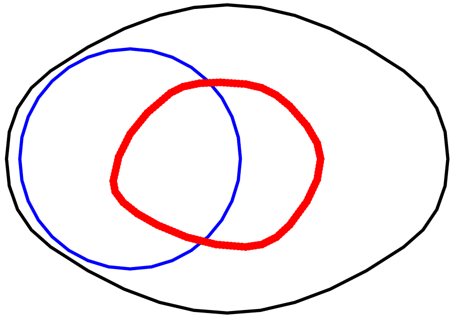
od;

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

$[y := 0]^6$

← $\{(x,?), (y,6), (z,2), (z,4)\}$

An Unsafe Solution



← $\{(x,?),(y,?),(z,?)\}$

$[\texttt{y := x}]^{1}$;

← $\{(x,?),(y,1),(z,?)\}$

$[\texttt{z := 1}]^{2}$;

← $\{(x,?),(y,1),(z,2),(y,5),(z,4)\}$

while $[\texttt{y} > \texttt{0}]^{3}$ **do**

← $\{(x,?),\boxed{(y,1)},\boxed{(z,2)},(y,5),(z,4)\}$

  $[\texttt{z := z} * \texttt{y}]^{4}$;

← $\{(x,?),\boxed{(y,1)},(y,5),(z,4)\}$

  $[\texttt{y := y} - \texttt{1}]^{5}$

← $\{(x,?),(y,5),(z,4)\}$

od;

← $\{(x,?),\boxed{(y,1)},\boxed{(z,2)},(y,5),(z,4)\}$

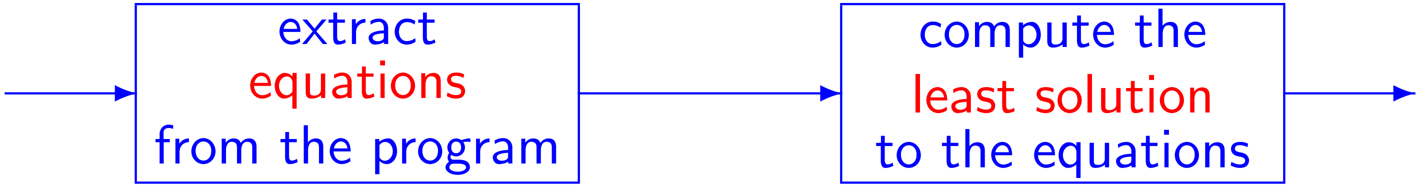
$[\texttt{y := 0}]^{6}$

← $\{(x,?),(y,6),\boxed{(z,2)},(z,4)\}$

Formalising the Development

Problem: which definitions reach which program points

The assignment $[x := a]^{\ell}$ reaches ℓ' if there is an execution where x was last assigned at ℓ


extract equations from the program

compute the least solution to the equations

- the programming language of interest
- abstract flow graphs
- extract and solve the equations

The WHILE *Language:*

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

Assignments and tests are (uniquely) labelled to allow the analyses to refer to these program fragments — the labels corresponds to pointers into the syntax tree.

We use abstract syntax and insert paranthesis to disambiguate the syntax — again this corresponds to thinking in terms of syntax trees.

OBS: the control structure is known — there is no need for a control flow analysis!

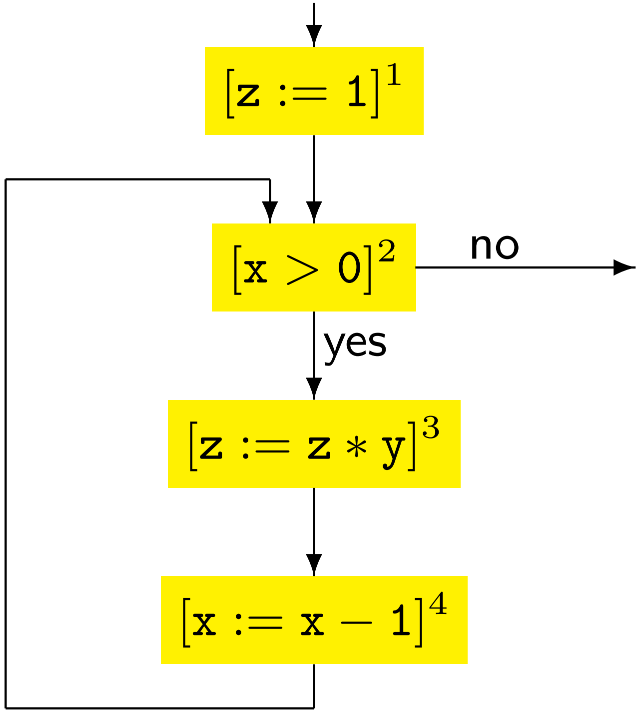
Talking about Flow Graphs

labels(S)	the set of nodes of the flow graphs of S
init(S)	the initial node of the flow graph of S; this is the unique node where the execution of the program starts
final(S)	the final nodes of the flow graphs for S; this is the set of nodes where the execution of the program may terminate
flow(S)	the edges of the flow graphs for S; there is an edge from one node to another if the flow of control may go from the first node to the second — used for forward analyses
$\text{flow}^{\text{R}}(S)$	the reversed edges of the flow graphs of S; there is an edge from one node to another if the flow of control may go from the second node to the first — used for backwards analyses

The Abstract Flow Graph

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

$$\begin{aligned}
\mathsf{labels}(\cdots) &= \{1,2,3,4\}\\
\mathsf{init}(\cdots) &= 1\\
\mathsf{final}(\cdots) &= \{2\}\\
\mathsf{flow}(\cdots) &= \{(1,2),(2,3),\\
&\quad (3,4),(4,2)\}\\
\mathsf{flow}^{\mathsf{R}}(\cdots) &= \{(2,1),(2,4),\\
&\quad (3,2),(4,3)\}
\end{aligned}$$



Computing the Information (1)

S	$\mathsf{labels}(S)$	$\mathsf{init}(S)$	$\mathsf{final}(S)$
$[x := a]^\ell$	$\{\ell\}$	ℓ	$\{\ell\}$
$[\texttt{skip}]^\ell$	$\{\ell\}$	ℓ	$\{\ell\}$
$S_1; S_2$	$\mathsf{labels}(S_1) \cup \mathsf{labels}(S_2)$	$\mathsf{init}(S_1)$	$\mathsf{final}(S_2)$
$\texttt{if}\ [b]^\ell\ \texttt{then}\ S_1$ $\texttt{else}\ S_2$	$\{\ell\} \cup \mathsf{labels}(S_1)$ $\cup\ \mathsf{labels}(S_2)$	ℓ	$\mathsf{final}(S_1)$ $\cup\ \mathsf{final}(S_2)$
$\texttt{while}\ [b]^\ell\ \texttt{do}\ S\ \texttt{od}$	$\{\ell\} \cup \mathsf{labels}(S)$	ℓ	$\{\ell\}$

Computing the Information (2)

S	$\mathsf{flow}(S)$	$\mathsf{blocks}(S)$
$[x := a]^\ell$	$\emptyset$	$\{[x := a]^\ell\}$
$[\texttt{skip}]^\ell$	$\emptyset$	$\{[\texttt{skip}]^\ell\}$
$S_1; S_2$	$\mathsf{flow}(S_1) \cup \mathsf{flow}(S_2)$ $\cup\{(\ell, \mathsf{init}(S_2)) \mid \ell \in \mathsf{final}(S_1)\}$	$\mathsf{blocks}(S_1)$ $\cup\ \mathsf{blocks}(S_2)$
$\texttt{if}\ [b]^\ell\ \texttt{then}\ S_1$ $\texttt{else}\ S_2$	$\mathsf{flow}(S_1) \cup \mathsf{flow}(S_2)$ $\cup\{(\ell, \mathsf{init}(S_1)), (\ell, \mathsf{init}(S_2))\}$	$\{[b]^\ell\} \cup \mathsf{blocks}(S_1)$ $\cup\ \mathsf{blocks}(S_2)$
$\texttt{while}\ [b]^\ell\ \texttt{do}\ S\ \texttt{od}$	$\{(\ell, \mathsf{init}(S))\} \cup \mathsf{flow}(S)$ $\cup\{(\ell', \ell) \mid \ell' \in \mathsf{final}(S)\}$	$\{[b]^\ell\} \cup \mathsf{blocks}(S)$

$$\mathsf{flow}^{\mathsf{R}}(S) = \{(\ell, \ell') \mid (\ell', \ell) \in \mathsf{flow}(S)\}$$

Simplifying assumptions

The program of interest $S_\star$ is often assumed to satisfy:

- $S_\star$ has isolated entries if there are no edges leading into $\text{init}(S_\star)$:

  $$\forall \ell : (\ell, \text{init}(S_\star)) \notin \text{flow}(S_\star)$$

- $S_\star$ has isolated exits if there are no edges leading out of labels in $\text{final}(S_\star)$:

  $$\forall \ell \in \text{final}(S_\star), \forall \ell' : (\ell, \ell') \notin \text{flow}(S_\star)$$

- $S_\star$ is label consistent if

  $$\forall [B_1^{\ell_1}], [B_2^{\ell_2}] \in \text{blocks}(S_\star) : \ell_1 = \ell_2 \Rightarrow B_1 = B_2$$

  This holds if $S_\star$ is uniquely labelled.

Reaching Definitions Analysis

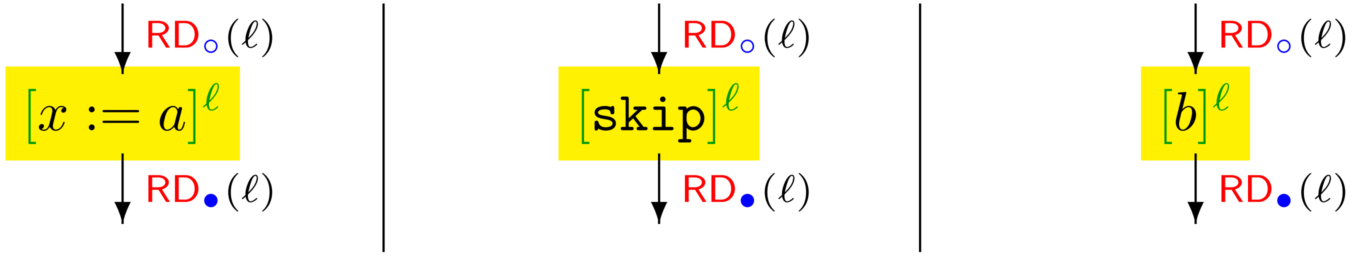
The aim of the Reaching Definitions Analysis is to determine

> For each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path.

Example

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

The Basic Idea



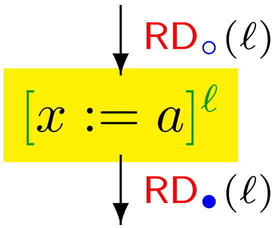
Analysis information:

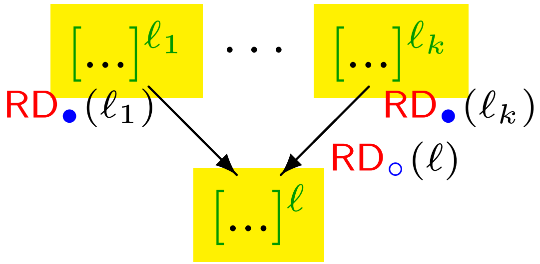
- $\mathsf{RD}_\circ(\ell)$: the definitions that reach the entry of block ℓ
- $\mathsf{RD}_\bullet(\ell)$: the definitions that reach the exit of block ℓ

Auxiliary information:

- $\mathsf{kill}_{\mathsf{RD}}(\ell)$: the definitions that are killed by an elementary block
- $\mathsf{gen}_{\mathsf{RD}}(\ell)$: the definitions that are generated by an elementary block

Analysis of the program





$$\mathsf{RD}_\bullet(\ell) \;=\; (\mathsf{RD}_\circ(\ell) \backslash \mathsf{kill}_{\mathsf{RD}}(B^\ell)) \cup \mathsf{gen}_{\mathsf{RD}}(B^\ell) \qquad \text{if } B^\ell \in \mathsf{blocks}(S_\star)$$

$$\mathsf{RD}_\circ(\ell) \;=\; \begin{cases} \{(x,?) \mid x \in FV(S_\star)\} & \text{if } \ell = \mathsf{init}(S_\star) \\ \bigcup\{\mathsf{RD}_\bullet(\ell') \mid (\ell',\ell) \in \mathsf{flow}(S_\star)\} & \text{otherwise} \end{cases}$$

$$\mathsf{kill}_{\mathsf{RD}}([x := a]^\ell) \;=\; \{(x,?)\} \cup \{(x,\ell') \mid B^{\ell'} \text{ is an assignment to } x\}$$

$$\mathsf{gen}_{\mathsf{RD}}([x := a]^\ell) \;=\; \{(x,\ell)\}$$

$$\mathsf{kill}_{\mathsf{RD}}(B^\ell) \;=\; \mathsf{gen}_{\mathsf{RD}}(B^\ell) \;=\; \emptyset \qquad \text{otherwise}$$

Example

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

Equations:

$$\begin{aligned}
\mathsf{RD}_\bullet(1) &= \mathsf{RD}_\circ(1) \setminus \{(\mathtt{y},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{y},6)\} \cup \{(\mathtt{y},1)\} & \mathsf{RD}_\circ(1) &= \{(\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?)\}\\
\mathsf{RD}_\bullet(2) &= \mathsf{RD}_\circ(2) \setminus \{(\mathtt{z},?),(\mathtt{z},2),(\mathtt{z},4)\} \cup \{(\mathtt{z},2)\} & \mathsf{RD}_\circ(2) &= \mathsf{RD}_\bullet(1)\\
\mathsf{RD}_\bullet(3) &= \mathsf{RD}_\circ(3) \setminus \emptyset \cup \emptyset & \mathsf{RD}_\circ(3) &= \mathsf{RD}_\bullet(2) \cup \mathsf{RD}_\bullet(5)\\
\mathsf{RD}_\bullet(4) &= \mathsf{RD}_\circ(4) \setminus \{(\mathtt{z},?),(\mathtt{z},2),(\mathtt{z},4)\} \cup \{(\mathtt{z},4)\} & \mathsf{RD}_\circ(4) &= \mathsf{RD}_\bullet(3)\\
\mathsf{RD}_\bullet(5) &= \mathsf{RD}_\circ(5) \setminus \{(\mathtt{y},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{y},6)\} \cup \{(\mathtt{y},5)\} & \mathsf{RD}_\circ(5) &= \mathsf{RD}_\bullet(4)\\
\mathsf{RD}_\bullet(6) &= \mathsf{RD}_\circ(6) \setminus \{(\mathtt{y},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{y},6)\} \cup \{(\mathtt{y},6)\} & \mathsf{RD}_\circ(6) &= \mathsf{RD}_\bullet(3)
\end{aligned}$$

Compute the least solution to these equations.

Solving the equations

```
1.  W := nil;
    for all $(\ell, \ell')$ in flow($S_\star$) do W := cons($(\ell, \ell')$,W);
    for all $\ell$ in labels($S_\star$) do
        if $\ell = \mathsf{init}(S_\star)$ then $\mathsf{RD}_\circ(\ell) := \{(x, ?) \mid x \in \mathsf{FV}(S_\star)\}$ else $\mathsf{RD}_\circ(\ell) := \emptyset$
2.  while W $\neq$ nil do
        $(\ell, \ell')$ := head(W); W := tail(W);
        if $\underbrace{(\mathsf{RD}_\circ(\ell) \setminus \mathsf{kill}_{\mathsf{RD}}(\ell)) \cup \mathsf{gen}_{\mathsf{RD}}(\ell)}_{\mathsf{RD}_\bullet(\ell)} \not\subseteq \mathsf{RD}_\circ(\ell')$
        then $\mathsf{RD}_\circ(\ell') := \mathsf{RD}_\circ(\ell') \cup \underbrace{(\mathsf{RD}_\circ(\ell) \setminus \mathsf{kill}_{\mathsf{RD}}(\ell)) \cup \mathsf{gen}_{\mathsf{RD}}(\ell)}_{\mathsf{RD}_\bullet(\ell)}$;
            for all $\ell''$ with $(\ell', \ell'')$ in flow($S_\star$) do W := cons($(\ell', \ell'')$,W);
```