Intra-procedural Data Flow Analysis

Introduction

Hanne Riis Nielson and Flemming Nielson

e-mail: \{riis,nielson\}@imm.dtu.dk
Informatics and Mathematical Modelling
Technical University of Denmark

©Hanne Riis Nielson & Flemming Nielson
Copyright

This set of transparencies is mainly based on

Flemming Nielson, Hanne Riis Nielson, Chris Hankin:
Principles of Program Analysis.

They may be used by instructors for presentations based on the book but may not be copied and distributed electronically or by other means.

A handout version with four slides per page may be used for producing paper copies of the slides for students; the students may be charged no fee beyond reasonable production cost.

The handout version of the transparencies is available from the webpage http://www.imm.dtu.dk/~riis/ppa.htm.
The Setting: Optimising Compilers

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
  - occasionally one optimise for space
  - not every optimisation succeeds in improving every program
  - occasionally an “optimisation” may slow down a program slightly but this is acceptable as long as it improves the average 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 independent

- 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

- **Front end**
- **Optimiser**
- **Back end**

- **Control flow analysis**
- **Data flow analysis**
- **Transformation**

**Aim:**
- To discover the hierarchical flow of control
- To determine information about data
- To modify the program
Example

(ack. Reinhard Wilhelm)

Algol-like arrays:

\[
\begin{align*}
1 & := 0; \\
\text{while } i \leq n \text{ do} & \\
\quad j & := 0; \\
\quad \text{while } j \leq m \text{ do} & \\
\quad & A[i,j] := B[i,j]; \\
\quad & j := j + 1 \\
\quad & \text{od; } \\
& i := i + 1 \\
\text{od}
\end{align*}
\]

C-like arrays:

\[
\begin{align*}
i & := 0; \\
\text{while } i \leq n \text{ do} & \\
\quad j & := 0; \\
\quad \text{while } j \leq m \text{ do} & \\
\quad & \text{temp := Base(A) + i * (m+1) + j; } \\
\quad & \text{Cont(temp) := Cont(Base(B) + i * (m+1) + j) + Cont(Base(C) + i * (m+1) + j; } \\
\quad & j := j + 1 \\
\quad & \text{od; } \\
& i := i + 1 \\
\text{od}
\end{align*}
\]
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

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

```
1 := 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

\[ i := 0; \]
\[ \text{while } i \leq n \text{ do} \]
\[ j := 0; \]
\[ \text{while } j \leq m \text{ do} \]
\[ t_1 := i \times (m+1) + j; \]
\[ \text{temp} := \text{Base}(A) + t_1; \]
\[ \text{Cont(temp)} := \text{Cont}(\text{Base}(B) + t_1) \]
\[ + \text{Cont}(\text{Base}(C) + t_1); \]
\[ j := j + 1 \]
\[ \text{od} \]
\[ i := i + 1 \]
\[ \text{od} \]

Loop invariant

Invariant code motion:

\[ t_2 := i \times (m+1); \]
\[ \text{while } j \leq m \text{ do} \]
\[ t_1 := t_2 + j; \]
\[ \text{temp} := \ldots \]
\[ \text{Cont(temp)} := \ldots \]
\[ j := \ldots \]
\[ \text{od} \]
Detection of Induction Variables

```
1 := 0;
while i <= n do
    j := 0;
    t2 := 1 * (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
```

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
```
Intra-procedural Data Flow Analysis

The Setting: Optimising Compilers — Example

Copy Analysis

```plaintext
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:

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

Intra-procedural Data Flow Analysis

The Setting: Optimising Compilers — Example


dead variable

1 := 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 + (n+1)
od

Dead code elimination:

1 := 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 + (n+1)
od
**Analyses and Transformations**

<table>
<thead>
<tr>
<th>Data Flow Analysis</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available expressions analysis</td>
<td>Common subexpression elimination</td>
</tr>
<tr>
<td>Detection of loop invariants</td>
<td>Invariant code motion</td>
</tr>
<tr>
<td>Detection of induction variables</td>
<td>Strength reduction</td>
</tr>
<tr>
<td>Copy analysis</td>
<td>Copy propagation</td>
</tr>
<tr>
<td>Live variables analysis</td>
<td>Dead code elimination</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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

Safe: faithful to the semantics

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

we cannot expect exact answers!
The Nature of Approximation

The exact world

Over-approximation

Under-approximation

Slogans: Err on the safe side!

Trade precision for efficiency!
Example

Program with labels for elementary blocks:

\[
\begin{align*}
   & \quad [y := x^1]; \\
   & \quad [z := 1^2]; \\
   & \quad \text{while } [y > 0]^3 \text{ do} \\
   & \quad \quad [z := z \cdot y]^4; \\
   & \quad \quad [y := y - 1]^5 \od; \\
   & \quad [y := 0]^6
\end{align*}
\]

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 $\ell'$ if there is an execution where $x$ was last assigned at $\ell$
Analysing the Program by Hand (1)

\[
\begin{align*}
[y := x] & \quad \{ (x,?), (y,?), (z,?) \} \\
[z := 1] & \quad \{ (x,?), (y,1), (z,?) \} \\
\textbf{while} \; [y > 0] & \; \textbf{do} \quad \{ (x,?), (y,1), (z,2) \} \\
[z := z * y] & \quad \{ (x,?), (y,1), (z,2) \} \\
[y := y - 1] & \quad \{ (x,?), (y,1), (z,2) \} \\
\textbf{od} & \\
[y := 0] & \quad \{ (x,?), (y,1), (z,2) \}
\end{align*}
\]
Analysing the Program by Hand (2)

\[ y := x^3; \]
\[ z := 1^2; \]
while \( y > 0 \) do
\[ y := y - 1 \]
\[ z := z \ast y^4; \]
\[ z := z \ast y \]
o; \]
\[ y := 0 \]
\[ y := \frac{x}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
\[ z := \frac{y}{?}; \]
Analysing the Program by Hand (3)

\[ y := x; \]
\[ z := 1; \]
\[ \text{while } y > 0 \text{ do} \]
\[ z := z \times y; \]
\[ y := y - 1; \]
\[ \text{od}; \]
\[ y := 0; \]

\[ \{ (x, ?), (y, ?), (z, ?) \} \]
\[ \{ (x, ?), (y, 1), (z, ?) \} \]
\[ \{ (x, ?), (y, 1), (z, 2), (y, 5), (z, 4) \} \]
\[ \{ (x, ?), (y, 1), (z, 2), (y, 5), (z, 4) \} \]
\[ \{ (x, ?), (y, 1), (z, 2), (y, 5), (z, 4) \} \]
\[ \{ (x, ?), (y, 5), (z, ?) \} \]
\[ \{ (x, ?), (y, 5), (z, ?) \} \]
\[ \{ (x, ?), (y, 1), (z, 2), (y, 5), (z, 4) \} \]
\[ \{ (x, ?), (y, 1), (z, 5), (z, 4) \} \]
\[ \{ (x, ?), (y, 1), (z, 2), (y, 5), (z, 4) \} \]
The Best Solution

\[ y := x^2; \quad \{ (x,?), (y,?), (z,?) \} \]
\[ z := 1^2; \quad \{ (x,?), (y,1), (z,?) \} \]
while \( y > 0 \) do
\[ z := z + y^2; \quad \{ (x,?), (y,1), (z,2), (y,5), (z,4) \} \]
\[ y := y - 1^5 \quad \{ (x,?), (y,1), (y,5), (z,4) \} \]
\[ \text{od;} \quad \{ (x,?), (y,5), (z,4) \} \]
\[ z := 0^6 \quad \{ (x,?), (y,1), (z,2), (y,5), (z,4) \} \]
\[ \text{\{ (x,?), (y,6), (z,2), (z,4) \}} \]
A Safe Solution — but not the Best

\[ y := x^1; \]
\[ z := 1^2; \]
while \( y > 0 \) do
\[ z := z + y^4; \]
\[ y := y - 1^5 \]
\[ y := 0^6 \]
\end{verbatim}
An Unsafe Solution

\[
\begin{align*}
[y := x]^1; & \quad \{(x, ?), (y, ?), (z, ?)\} \\
[z := 1]^2; & \quad \{(x, ?), (y, 1), (z, ?)\} \\
\text{while } [y > 0]^3 & \text{ do} \quad \{(x, ?), (y, 1), (z, 2), (y, 5), (z, 4)\} \\
[z := z + y]^4; & \quad \{(x, ?), (y, 1), (z, 2), (y, 5), (z, 4)\} \\
[y := y - 1]^5 & \quad \{(x, ?), (y, 5), (z, 4)\} \\
\text{od;} & \quad \{(x, ?), (y, 5), (z, 4)\} \\
[y := 0]^6 & \quad \{(x, ?), (y, 6), (z, 2), (z, 4)\}
\end{align*}
\]
Formalising the Development

Problem: which definitions reach which program points

The assignment \([x := a]\) reaches \(\ell'\) if there is an execution where \(x\) was last assigned at \(\ell\)

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

extract equations from the program

compute the least solution to the equations
The **While Language:**

\[ a ::= x \mid n \mid a_1 \text{op}_a a_2 \]

\[ b ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{op}_b b_2 \mid a_1 \text{op}_r a_2 \]

\[ S ::= \{x := a\}^\ell \mid \{\text{skip}\}^\ell \mid S_1; S_2 \mid \]

\[ \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \mid \text{while } [b]^\ell \text{ do } S \text{ od} \]

Assignments and tests are (uniquely) labelled to allow the analyses to refer to these program fragments — the labels correspond 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**

- $\text{labels}(S)$: the set of nodes of the flow graphs of $S$
- $\text{init}(S)$: the initial node of the flow graph of $S$; this is the unique node where the execution of the program starts
- $\text{final}(S)$: the final nodes of the flow graphs for $S$; this is the set of nodes where the execution of the program may terminate
- $\text{flow}(S)$: 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
- $\text{flow}^R(S)$: 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
\textbf{The Abstract Flow Graph}

\textbf{Example:} $z:=1^1; \text{ while } x>0^2 \text{ do } z:=z*y^3; x:=x-1^4 \text{ od}$

- $\text{labels(⋯)} = \{1, 2, 3, 4\}$
- $\text{init(⋯)} = 1$
- $\text{final(⋯)} = \{2\}$
- $\text{flow(⋯)} = \{(1, 2), (2, 3), (3, 4), (4, 2)\}$
- $\text{flow}^R(⋯) = \{(2, 1), (2, 4), (3, 2), (4, 3)\}$
## Computing the Information (1)

<table>
<thead>
<tr>
<th>$S$</th>
<th>labels($S$)</th>
<th>init($S$)</th>
<th>final($S$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[x := a]^{\ell}$</td>
<td>${\ell}$</td>
<td>$\ell$</td>
<td>${\ell}$</td>
</tr>
<tr>
<td>$[\text{skip}]^{\ell}$</td>
<td>${\ell}$</td>
<td>$\ell$</td>
<td>${\ell}$</td>
</tr>
<tr>
<td>$S_1; S_2$</td>
<td>labels($S_1$) $\cup$ labels($S_2$)</td>
<td>init($S_1$)</td>
<td>final($S_2$)</td>
</tr>
<tr>
<td>if $[b]^{\ell}$ then $S_1$ else $S_2$</td>
<td>${\ell} \cup$ labels($S_1$) $\cup$ labels($S_2$)</td>
<td>$\ell$</td>
<td>final($S_1$) $\cup$ final($S_2$)</td>
</tr>
<tr>
<td>while $[b]^{\ell}$ do $S$ od</td>
<td>${\ell} \cup$ labels($S$)</td>
<td>$\ell$</td>
<td>${\ell}$</td>
</tr>
</tbody>
</table>
Computing the Information (2)

\[
\begin{array}{|c|c|c|}
\hline
S & \text{flow}(S) & \text{blocks}(S) \\
\hline
[x := a]^\ell & \emptyset & \{[x := a]^\ell\} \\
[\text{skip}]^\ell & \emptyset & \{[\text{skip}]^\ell\} \\
S_1; S_2 & \text{flow}(S_1) \cup \text{flow}(S_2) & \text{blocks}(S_1) \\
& \cup \{(\ell, \text{init}(S_2)) | \ell \in \text{final}(S_1)\} & \cup \text{blocks}(S_2) \\
\text{if } [b]^\ell \text{ then } S_1 & \text{flow}(S_1) \cup \text{flow}(S_2) & \{[b]^\ell\} \cup \text{blocks}(S_1) \\
\text{else } S_2 & \cup \{(\ell, \text{init}(S_1)), (\ell, \text{init}(S_2))\} & \cup \text{blocks}(S_2) \\
\text{while } [b]^\ell \text{ do } S \text{ od} & \{(\ell, \text{init}(S))\} \cup \text{flow}(S) & \{[b]^\ell\} \cup \text{blocks}(S) \\
& \cup \{(\ell', \ell) | \ell' \in \text{final}(S)\} & \\
\hline
\end{array}
\]

\[\text{flow}^R(S) = \{(\ell, \ell') | (\ell', \ell) \in \text{flow}(S)\}\]
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
  \[ \forall [B_1], [B_2] \in \text{blocks}(S_\star) : \ell_1 = \ell_2 \Rightarrow B_1 = B_2 \]
  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
\[
\begin{align*}
y & := x^1; \\
z & := 1^2; \\
\text{while } & y > 0 \text{ do } \begin{cases} z := z + y^4; \\ y := y - 1 \end{cases} \text{ od; } y := 0^6 \end{align*}
\]
The Basic Idea

Analysis information:
- $\text{RD}_e(\ell)$: the definitions that reach the entry of block $\ell$
- $\text{RD}_e(\ell)$: the definitions that reach the exit of block $\ell$

Auxiliary information:
- $\text{kill}_{\text{RD}}(\ell)$: the definitions that are killed by an elementary block
- $\text{gen}_{\text{RD}}(\ell)$: the definitions that are generated by an elementary block
Analysis of the program

\[
\begin{align*}
RD_\bullet(\ell) &= (RD_\circ(\ell) \setminus kill_{RD}(B^{\ell})) \cup gen_{RD}(B^{\ell}) & \text{if } B^{\ell} \in \text{blocks}(S_\circ) \\
RD_\circ(\ell) &= \begin{cases} 
\{(x,?) \mid x \in FV(S_\circ)\} & \text{if } \ell = \text{init}(S_\circ) \\
\bigcup\{RD_\circ(\ell') \mid (\ell', \ell) \in \text{flow}(S_\circ)\} & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{kill}_{RD}([x := a]^\ell) &= \{(x,?)\} \cup \{(x, \ell') \mid B^{\ell'} \text{ is an assignment to } x\} \\
\text{gen}_{RD}([x := a]^\ell) &= \{(x, \ell)\} \\
\text{kill}_{RD}(B^{\ell}) &= \text{gen}_{RD}(B^{\ell}) = \emptyset & \text{otherwise}
\end{align*}
\]
Example

\[y := x; \quad z := 1; \quad \text{while } y > 0 \text{ do } z := z \times y; \quad [y := y - 1] \quad \text{od}; \quad [y := 0]\]

Equations:

\[RD^*_1(1) = RD^*_1(1) \setminus \{(y, ?), (y, 1), (y, 5), (y, 6)\} \cup \{(y, 1)\}\]
\[RD^*_2(2) = RD^*_2(2) \setminus \{(z, ?), (z, 2), (z, 4)\} \cup \{(z, 2)\}\]
\[RD^*_3(3) = RD^*_3(3) \setminus \emptyset\]
\[RD^*_4(4) = RD^*_4(4) \setminus \{(z, ?), (z, 2), (z, 4)\} \cup \{(z, 4)\}\]
\[RD^*_5(5) = RD^*_5(5) \setminus \{(y, ?), (y, 1), (y, 5), (y, 6)\} \cup \{(y, 5)\}\]
\[RD^*_6(6) = RD^*_6(6) \setminus \{(y, ?), (y, 1), (y, 5), (y, 6)\} \cup \{(y, 6)\}\]

Compute the least solution to these equations.
**Solving the equations**

1. \[ W := \text{nil}; \]
   
   for all \((\ell, \ell')\) in \(\text{flow}(S_*)\) do \(W := \text{cons}((\ell, \ell'), W);\)
   
   for all \(\ell\) in \(\text{labels}(S_*)\) do
     
     if \(\ell = \text{init}(S_*)\) then \(\text{RD}_c(\ell) := \{(x, ?) \mid x \in \text{FV}(S_*)\}\) else \(\text{RD}_c(\ell) := \emptyset\)

2. while \(W \neq \text{nil}\) do
   
   \((\ell, \ell') := \text{head}(W); W := \text{tail}(W);\)
   
   \[
   \text{if } (\text{RD}_c(\ell) \setminus \text{kill}_{\text{RD}}(\ell)) \cup \text{gen}_{\text{RD}}(\ell) \subseteq \text{RD}_c(\ell')
   \]
   
   then \(\text{RD}_c(\ell') := \text{RD}_c(\ell') \cup (\text{RD}_c(\ell) \setminus \text{kill}_{\text{RD}}(\ell)) \cup \text{gen}_{\text{RD}}(\ell);\)

   for all \(\ell''\) with \((\ell', \ell'')\) in \(\text{flow}(S_*)\) do \(W := \text{cons}((\ell', \ell''), W);\)