Program Analysis (470)

Herbert Wiklicky
herbert@doc.ic.ac.uk

Department of Computing
Imperial College London

Spring 2015

Program Analysis

Program analysis is an automated technique for finding out properties of programs without having to execute them.

Static Analysis vs **Dynamic Testing**

- Compiler Optimisation
- Program Verification
- Security Analysis

Program Analysis

Program analysis is an automated technique for finding out properties of programs without having to execute them.

Static Analysis vs **Dynamic Testing**

- Compiler Optimisation
- Program Verification
- Security Analysis

Program Analysis

Program analysis is an automated technique for finding out properties of programs without having to execute them.

Static Analysis vs **Dynamic Testing**

- Compiler Optimisation
- Program Verification
- Security Analysis

Program Analysis

Program analysis is an automated technique for finding out properties of programs without having to execute them.

Static Analysis vs **Dynamic Testing**

- Compiler Optimisation
- Program Verification
- Security Analysis

Program Analysis

Program analysis is an automated technique for finding out properties of programs without having to execute them.

Static Analysis vs **Dynamic Testing**

- Compiler Optimisation
- Program Verification
- Security Analysis

Techniques

Some techniques used in program analysis include:

- Data Flow Analysis
- Control Flow Analysis
- Types and Effects Systems
- Abstract Interpretation

Flemming Nielson, Hanne Riis Nielson and Chris Hankin:
Principles of Program Analysis. Springer Verlag, 1999/2005.

Techniques

Some techniques used in program analysis include:

- Data Flow Analysis
- Control Flow Analysis
- Types and Effects Systems
- Abstract Interpretation

Flemming Nielson, Hanne Riis Nielson and Chris Hankin:
Principles of Program Analysis. Springer Verlag, 1999/2005.

Techniques

Some techniques used in program analysis include:

- Data Flow Analysis
- Control Flow Analysis
- Types and Effects Systems
- Abstract Interpretation

Flemming Nielson, Hanne Riis Nielson and Chris Hankin:
Principles of Program Analysis. Springer Verlag, 1999/2005.

Techniques

Some techniques used in program analysis include:

- Data Flow Analysis
- Control Flow Analysis
- Types and Effects Systems
- Abstract Interpretation

Flemming Nielson, Hanne Riis Nielson and Chris Hankin: *Principles of Program Analysis*. Springer Verlag, 1999/2005.

Techniques

Some techniques used in program analysis include:

- Data Flow Analysis
- Control Flow Analysis
- Types and Effects Systems
- Abstract Interpretation

Flemming Nielson, Hanne Riis Nielson and Chris Hankin:
Principles of Program Analysis. Springer Verlag, 1999/2005.

Techniques

Some techniques used in program analysis include:

- Data Flow Analysis
- Control Flow Analysis
- Types and Effects Systems
- Abstract Interpretation

Flemming Nielson, Hanne Riis Nielson and Chris Hankin:
Principles of Program Analysis. Springer Verlag, 1999/2005.

A First Example

Consider the following fragment in *some* procedural language.

```
$m \leftarrow 2$;
while $n > 1$ do
    $m \leftarrow m \times n$;
    $n \leftarrow n - 1$
end while
stop
```

```
$[m \leftarrow 2]^1$;
while $[n > 1]^2$ do
    $[m \leftarrow m \times n]^3$;
    $[n \leftarrow n - 1]^4$
end while
$[\text{stop}]^5$
```

We annotate a program such that it becomes clear about what program point we are talking about.

A First Example

Consider the following fragment in *some* procedural language.

```
$m \leftarrow 2$;
while $n > 1$ do
    $m \leftarrow m \times n$;
    $n \leftarrow n - 1$
end while
stop
```

```
$[m \leftarrow 2]^1$;
while $[n > 1]^2$ do
    $[m \leftarrow m \times n]^3$;
    $[n \leftarrow n - 1]^4$
end while
$[\text{stop}]^5$
```

We annotate a program such that it becomes clear about what program point we are talking about.

A First Example

Consider the following fragment in *some* procedural language.

```
$m \leftarrow 2$;
while $n > 1$ do
    $m \leftarrow m \times n$;
    $n \leftarrow n - 1$
end while
stop
```

```
$[m \leftarrow 2]^1$;
while $[n > 1]^2$ do
    $[m \leftarrow m \times n]^3$;
    $[n \leftarrow n - 1]^4$
end while
$[\textbf{stop}]^5$
```

We annotate a program such that it becomes clear about what program point we are talking about.

A Parity Analysis

Claim: This program fragment always returns an **even** m, idependently of the initial values of m and n.

We can statically determine that in any circumstances the value of m at the last statement will be **even** for any input n.

A program analysis, so-called parity analysis, can determine this by propagating the even/odd or *parity* information *forwards* form the start of the program.

A Parity Analysis

Claim: This program fragment always returns an **even** m, idependently of the initial values of m and n**.**

We can statically determine that in any circumstances the value of m at the last statement will be **even** for any input n**.**

A program analysis, so-called parity analysis, can determine this by propagating the even/odd or *parity* information *forwards* form the start of the program.

A Parity Analysis

Claim: This program fragment always returns an **even** m, idependently of the initial values of m and n.

We can statically determine that in any circumstances the value of m at the last statement will be **even** for any input n.

A program analysis, so-called parity analysis, can determine this by propagating the even/odd or *parity* information *forwards* form the start of the program.

Properties

We will assign to each variable one of three properties:

- **even** — the value is known to be even
- **odd** — the value is known to be odd
- **unknown** — the parity of the value is unknown

For both variables m and n we record its parity at each stage of the computation (beginning of each statement).

Properties

We will assign to each variable one of three properties:

- **even** — the value is known to be even
- **odd** — the value is known to be odd
- **unknown** — the parity of the value is unknown

For both variables m and n we record its parity at each stage of the computation (beginning of each statement).

Properties

We will assign to each variable one of three properties:

- **even** — the value is known to be even
- **odd** — the value is known to be odd
- **unknown** — the parity of the value is unknown

For both variables m and n we record its parity at each stage of the computation (beginning of each statement).

Properties

We will assign to each variable one of three properties:

- **even** — the value is known to be even
- **odd** — the value is known to be odd
- **unknown** — the parity of the value is unknown

For both variables m and n we record its parity at each stage of the computation (beginning of each statement).

Properties

We will assign to each variable one of three properties:

- **even** — the value is known to be even
- **odd** — the value is known to be odd
- **unknown** — the parity of the value is unknown

For both variables m and n we record its parity at each stage of the computation (beginning of each statement).

A First Example

Executing the program with *abstract* values, parity, for m and n.

```
$m \leftarrow 2$;
while $n > 1$ do
    $m \leftarrow m \times n$;
    $n \leftarrow n - 1$
end while
stop
```

Important: We can restart the loop!

A First Example

Executing the program with *abstract* values, parity, for m and n.

```
$m \leftarrow 2$;                    ▷ unknown(m) – unknown(n)
while $n > 1$ do
    $m \leftarrow m \times n$;
    $n \leftarrow n - 1$
end while
stop
```

Important: We can restart the loop!

A First Example

Executing the program with *abstract* values, parity, for m and n.

```
$m \leftarrow 2$;                          ▷ unknown(m) – unknown(n)
while $n > 1$ do                           ▷ even(m) – unknown(n)
    $m \leftarrow m \times n$;
    $n \leftarrow n - 1$
end while
stop
```

Important: We can restart the loop!

A First Example

Executing the program with *abstract* values, parity, for m and n.

```
$m \leftarrow 2$;                              ▷ unknown(m) – unknown(n)
while $n > 1$ do                                   ▷ even(m) – unknown(n)
    $m \leftarrow m \times n$;                     ▷ even(m) – unknown(n)
    $n \leftarrow n - 1$
end while
stop                                              ▷ even(m) – unknown(n)
```

Important: We can restart the loop!

A First Example

Executing the program with *abstract* values, parity, for m and n.

```
$m \leftarrow 2$;                    ▷ unknown(m) – unknown(n)
while $n > 1$ do                     ▷ even(m) – unknown(n)
    $m \leftarrow m \times n$;       ▷ even(m) – unknown(n)
    $n \leftarrow n - 1$             ▷ even(m) – unknown(n)
end while
stop                                 ▷ even(m) – unknown(n)
```

Important: We can restart the loop!

A First Example

Executing the program with *abstract* values, parity, for m and n.

```
$m \leftarrow 2$;                  ▷ unknown(m) – unknown(n)
while $n > 1$ do                   ▷ even(m) – unknown(n)
    $m \leftarrow m \times n$;     ▷ even(m) – unknown(n)
    $n \leftarrow n - 1$           ▷ even(m) – unknown(n)
end while                          ▷ even(m) – unknown(n)
stop                               ▷ even(m) – unknown(n)
```

Important: We can restart the loop!

A First Example

Executing the program with *abstract* values, parity, for m and n.

```
$m \leftarrow 2$;                      ▷ unknown(m) – unknown(n)
while $n > 1$ do                       ▷ even(m) – unknown(n)
    $m \leftarrow m \times n$;
    $n \leftarrow n - 1$
end while
stop                                   ▷ even(m) – unknown(n)
```

Important: We can restart the loop!

A First Example

Executing the program with *abstract* values, parity, for m and n.

```
$m \leftarrow 2$;                 ▷ unknown(m) – unknown(n)
while $n > 1$ do                  ▷ even(m) – unknown(n)
    $m \leftarrow m \times n$;    ▷ even(m) – unknown(n)
    $n \leftarrow n - 1$
end while
stop                              ▷ even(m) – unknown(n)
```

Important: We can restart the loop!

A First Example

Executing the program with *abstract* values, parity, for m and n.

```
$m \leftarrow 2$;                ▷ unknown(m) – unknown(n)
while $n > 1$ do                 ▷ even(m) – unknown(n)
    $m \leftarrow m \times n$;   ▷ even(m) – unknown(n)
    $n \leftarrow n - 1$         ▷ even(m) – unknown(n)
end while
stop                             ▷ even(m) – unknown(n)
```

Important: We can restart the loop!

A First Example

Executing the program with *abstract* values, parity, for m and n.

```
$m \leftarrow 2$;                                  ▷ unknown(m) – unknown(n)
while $n > 1$ do                                   ▷ even(m) – unknown(n)
    $m \leftarrow m \times n$;                     ▷ even(m) – unknown(n)
    $n \leftarrow n - 1$                           ▷ even(m) – unknown(n)
end while                                          ▷ even(m) – unknown(n)
stop                                               ▷ even(m) – unknown(n)
```

Important: We can restart the loop!

A First Example

Executing the program with *abstract* values, parity, for m and n.

```
$m \leftarrow 2$;                  ▷ unknown(m) – unknown(n)
while $n > 1$ do                   ▷ even(m) – unknown(n)
    $m \leftarrow m \times n$;     ▷ even(m) – unknown(n)
    $n \leftarrow n - 1$           ▷ even(m) – unknown(n)
end while                          ▷ even(m) – unknown(n)
stop                               ▷ even(m) – unknown(n)
```

Important: We can restart the loop!

A First Example

Executing the program with *abstract* values, parity, for m and n.

```
$m \leftarrow 2$;                    ▷ unknown(m) – unknown(n)
while $n > 1$ do                     ▷ even(m) – unknown(n)
    $m \leftarrow m \times n$;       ▷ even(m) – unknown(n)
    $n \leftarrow n - 1$             ▷ even(m) – unknown(n)
end while                            ▷ even(m) – unknown(n)
stop                                 ▷ even(m) – unknown(n)
```

Important: We can restart the loop!

A First Example

The first program computes 2 times the factorial for any positive value of n. Replacing ‘2’ by ‘1’ in the first statement gives:

```
$m \leftarrow 1$;
while $n > 1$ do
    $m \leftarrow m \times n$;
    $n \leftarrow n - 1$
end while
stop
```

i.e. the factorial – but then the program analysis is unable to tell us anything about the parity of m at the end of the execution.

A First Example

The first program computes 2 times the factorial for any positive value of n. Replacing ‘2’ by ‘1’ in the first statement gives:

```
m ← 1;                        ▷ unknown(m) – unknown(n)
while n > 1 do
    m ← m × n;
    n ← n − 1
end while
stop
```

i.e. the factorial – but then the program analysis is unable to tell us anything about the parity of m at the end of the execution.

A First Example

The first program computes 2 times the factorial for any positive value of n. Replacing ‘2’ by ‘1’ in the first statement gives:

```
$m \leftarrow 1$;                          ▷ unknown(m) – unknown(n)
while $n > 1$ do                           ▷ odd(m) – unknown(n)
    $m \leftarrow m \times n$;
    $n \leftarrow n - 1$
end while
stop
```

i.e. the factorial – but then the program analysis is unable to tell us anything about the parity of m at the end of the execution.

A First Example

The first program computes 2 times the factorial for any positive value of n. Replacing ‘2’ by ‘1’ in the first statement gives:

```
$m \leftarrow 1$;                    ▷ unknown(m) – unknown(n)
while $n > 1$ do                     ▷ odd(m) – unknown(n)
    $m \leftarrow m \times n$;       ▷ odd(m) – unknown(n)
    $n \leftarrow n - 1$
end while
stop                                 ▷ odd(m) – unknown(n)
```

i.e. the factorial – but then the program analysis is unable to tell us anything about the parity of m at the end of the execution.

A First Example

The first program computes 2 times the factorial for any positive value of n. Replacing ‘2’ by ‘1’ in the first statement gives:

```
$m \leftarrow 1$;                    ▷ unknown(m) – unknown(n)
while $n > 1$ do                     ▷ odd(m) – unknown(n)
    $m \leftarrow m \times n$;       ▷ odd(m) – unknown(n)
    $n \leftarrow n - 1$             ▷ unknown(m) – unknown(n)
end while
stop                                 ▷ odd(m) – unknown(n)
```

i.e. the factorial – but then the program analysis is unable to tell us anything about the parity of m at the end of the execution.

A First Example

The first program computes 2 times the factorial for any positive value of n. Replacing ‘2’ by ‘1’ in the first statement gives:

```
$m \leftarrow 1$;                   ▷ unknown(m) – unknown(n)
while $n > 1$ do                    ▷ odd(m) – unknown(n)
    $m \leftarrow m \times n$;      ▷ odd(m) – unknown(n)
    $n \leftarrow n - 1$            ▷ unknown(m) – unknown(n)
end while                           ▷ unknown(m) – unknown(n)
stop                                ▷ odd(m) – unknown(n)
```

i.e. the factorial – but then the program analysis is unable to tell us anything about the parity of m at the end of the execution.

A First Example

The first program computes 2 times the factorial for any positive value of n. Replacing ‘2’ by ‘1’ in the first statement gives:

```
$m \leftarrow 1$;                ▷ unknown(m) – unknown(n)
while $n > 1$ do                 ▷ unknown(m) – unknown(n)
    $m \leftarrow m \times n$;
    $n \leftarrow n - 1$
end while
stop                             ▷ odd(m) – unknown(n)
```

i.e. the factorial – but then the program analysis is unable to tell us anything about the parity of m at the end of the execution.

A First Example

The first program computes 2 times the factorial for any positive value of n. Replacing ‘2’ by ‘1’ in the first statement gives:

```
$m \leftarrow 1$;                      ▷ unknown(m) – unknown(n)
while $n > 1$ do                       ▷ unknown(m) – unknown(n)
    $m \leftarrow m \times n$;         ▷ unknown(m) – unknown(n)
    $n \leftarrow n - 1$
end while
stop                                   ▷ unknown(m) – unknown(n)
```

i.e. the factorial – but then the program analysis is unable to tell us anything about the parity of m at the end of the execution.

A First Example

The first program computes 2 times the factorial for any positive value of n. Replacing ‘2’ by ‘1’ in the first statement gives:

```
$m \leftarrow 1$;                  ▷ unknown(m) – unknown(n)
while $n > 1$ do                  ▷ unknown(m) – unknown(n)
    $m \leftarrow m \times n$;     ▷ unknown(m) – unknown(n)
    $n \leftarrow n - 1$
end while
stop                              ▷ unknown(m) – unknown(n)
```

i.e. the factorial – but then the program analysis is unable to tell us anything about the parity of m at the end of the execution.

Loss of Precision

The analysis of the new program does not give a satisfying result because:

- m could be **even** — if the input $n > 1$, or
- m could be **odd** — if the input $n \leq 1$.

However, even if we fix/require the input to be positive and **even** — e.g. by some suitable conditional assignment — the program analysis still might not be able to accurately predict that m will be **even** at statement **5**.

Loss of Precision

The analysis of the new program does not give a satisfying result because:

- m could be **even** — if the input $n > 1$, or
- m could be **odd** — if the input $n \leq 1$.

However, even if we fix/require the input to be positive and **even** — e.g. by some suitable conditional assignment — the program analysis still might not be able to accurately predict that m will be **even** at statement **5**.

Loss of Precision

The analysis of the new program does not give a satisfying result because:

- m could be **even** — if the input $n > 1$, or
- m could be **odd** — if the input $n \leq 1$.

However, even if we fix/require the input to be positive and **even** — e.g. by some suitable conditional assignment — the program analysis still might not be able to accurately predict that m will be **even** at statement **5**.

Loss of Precision

The analysis of the new program does not give a satisfying result because:

- m could be **even** — if the input $n > 1$, or
- m could be **odd** — if the input $n \leq 1$.

However, even if we fix/require the input to be positive and **even** — e.g. by some suitable conditional assignment — the program analysis still might not be able to accurately predict that m will be **even** at statement **5**.

Safe Approximations

Such a loss of precession is a common feature of program analysis: many properties that we are interested in are essentially undecidable and therefore we cannot hope to detect (all of) them accurately.

We only aim to ensure that the answers/results we obtain by program analysis are at least safe, i.e.

- yes means *definitely* yes,
- no means *maybe* no.

Safe Approximations

Such a loss of precession is a common feature of program analysis: many properties that we are interested in are essentially undecidable and therefore we cannot hope to detect (all of) them accurately.

We only aim to ensure that the answers/results we obtain by program analysis are at least safe, i.e.

- yes means *definitely* yes,
- no means *maybe* no.

Safe Approximations

Such a loss of precession is a common feature of program analysis: many properties that we are interested in are essentially undecidable and therefore we cannot hope to detect (all of) them accurately.

We only aim to ensure that the answers/results we obtain by program analysis are at least safe, i.e.

- yes means *definitely* yes,
- no means *maybe* no.

Safe Approximations

Such a loss of precession is a common feature of program analysis: many properties that we are interested in are essentially undecidable and therefore we cannot hope to detect (all of) them accurately.

We only aim to ensure that the answers/results we obtain by program analysis are at least safe, i.e.

- yes means *definitely* yes,
- no means *maybe* no.

Facets of Program Analysis

We can identify the following facets of program analysis which we will mostely cover in this course:

- specification,
- implementation,
- correctness,
- *applications*.

Facets of Program Analysis

We can identify the following facets of program analysis which we will mostely cover in this course:

- specification,
- implementation,
- correctness,
- *applications*.

Facets of Program Analysis

We can identify the following facets of program analysis which we will mostely cover in this course:

- specification,
- implementation,
- correctness,
- *applications*.

Facets of Program Analysis

We can identify the following facets of program analysis which we will mostely cover in this course:

- specification,
- implementation,
- correctness,
- *applications*.

Facets of Program Analysis

We can identify the following facets of program analysis which we will mostely cover in this course:

- specification,
- implementation,
- correctness,
- *applications*.

Data Flow Analysis

The starting point for data flow analysis is a representation of the control flow graph of the program: the nodes of such a graph may represent individual statements – as in a flowchart – or sequences of statements; arcs specify how control may be passed during program execution.

The data flow analysis is usually specified as a set of equations which associate analysis information with program points which correspond to the nodes in the control flow graph. This information may be propagated *forwards* through the program (e.g. parity analysis) or *backwards*.

When the control flow graph is not explicitly given, we need a preliminary control flow analysis

Data Flow Analysis

The starting point for data flow analysis is a representation of the control flow graph of the program: the nodes of such a graph may represent individual statements – as in a flowchart – or sequences of statements; arcs specify how control may be passed during program execution.

The data flow analysis is usually specified as a set of equations which associate analysis information with program points which correspond to the nodes in the control flow graph. This information may be propagated *forwards* through the program (e.g. parity analysis) or *backwards*.

When the control flow graph is not explicitly given, we need a preliminary control flow analysis

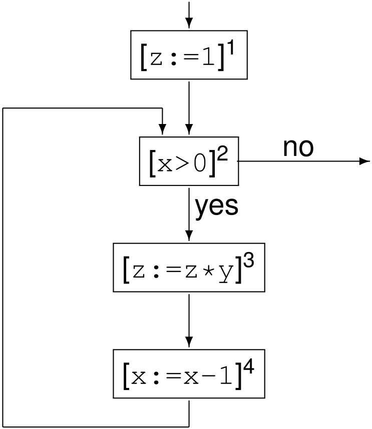
Data Flow Analysis

The starting point for data flow analysis is a representation of the control flow graph of the program: the nodes of such a graph may represent individual statements – as in a flowchart – or sequences of statements; arcs specify how control may be passed during program execution.

The data flow analysis is usually specified as a set of equations which associate analysis information with program points which correspond to the nodes in the control flow graph. This information may be propagated *forwards* through the program (e.g. parity analysis) or *backwards*.

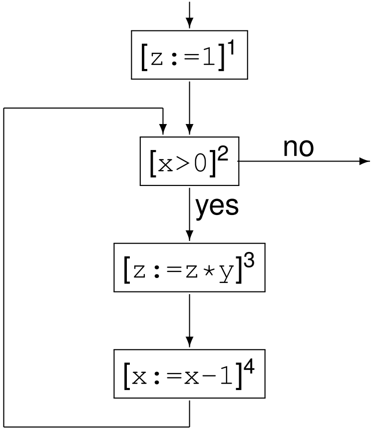
When the control flow graph is not explicitly given, we need a preliminary control flow analysis

Control Flow Information



This allows us to determine the predecessors *pred* and successors *succ* of each statement, e.g. $\mathit{pred}(2) = \{1, 4\}$.

Control Flow Information



This allows us to determine the predecessors *pred* and successors *succ* of each statement, e.g. $\mathit{pred}(2) = \{1, 4\}$.

Reaching Definition

Reaching Definition (*RD*) analysis determines which set of definitions (i.e. assignments) are current when control reaches a certain program point *p*.

The analysis can be specified by equations of the form:

$$\mathrm{RD}_{entry}(p) = \begin{cases} \mathrm{RD}_{init} & \text{if } p \text{ is initial} \\ \bigcup\limits_{p' \in pred(p)} \mathrm{RD}_{exit}(p') & \text{otherwise} \end{cases}$$

$$\mathrm{RD}_{exit}(p) = (\mathrm{RD}_{entry}(p) \setminus kill_{\mathrm{RD}}(p)) \cup gen_{\mathrm{RD}}(p)$$

Reaching Definition

Reaching Definition (*RD*) analysis determines which set of definitions (i.e. assignments) are current when control reaches a certain program point *p*.

The analysis can be specified by equations of the form:

$$\mathrm{RD}_{entry}(p) = \begin{cases} \mathrm{RD}_{init} & \text{if } p \text{ is initial} \\ \bigcup\limits_{p' \in pred(p)} \mathrm{RD}_{exit}(p') & \text{otherwise} \end{cases}$$

$$\mathrm{RD}_{exit}(p) = (\mathrm{RD}_{entry}(p) \setminus kill_{\mathrm{RD}}(p)) \cup gen_{\mathrm{RD}}(p)$$

Reaching Definition

Reaching Definition (*RD*) analysis determines which set of definitions (i.e. assignments) are current when control reaches a certain program point p.

The analysis can be specified by equations of the form:

$$\mathrm{RD}_{entry}(p) = \begin{cases} \mathrm{RD}_{init} & \text{if } p \text{ is initial} \\ \displaystyle\bigcup_{p' \in pred(p)} \mathrm{RD}_{exit}(p') & \text{otherwise} \end{cases}$$

$$\mathrm{RD}_{exit}(p) = (\mathrm{RD}_{entry}(p) \setminus kill_{\mathrm{RD}}(p)) \cup gen_{\mathrm{RD}}(p)$$

Reaching Definition

Reaching Definition (*RD*) analysis determines which set of definitions (i.e. assignments) are current when control reaches a certain program point p.

The analysis can be specified by equations of the form:

$$\mathrm{RD}_{entry}(p) = \begin{cases} \mathrm{RD}_{init} & \text{if } p \text{ is initial} \\ \displaystyle\bigcup_{p' \in pred(p)} \mathrm{RD}_{exit}(p') & \text{otherwise} \end{cases}$$

$$\mathrm{RD}_{exit}(p) = (\mathrm{RD}_{entry}(p) \backslash kill_{\mathrm{RD}}(p)) \cup gen_{\mathrm{RD}}(p)$$

Analysis Information

At each program point some definitions get "killed" (those which define the same variable as at the program point) while others are "generated".

A suitable representation for properties are sets of pairs, where each pair contains a variable x and a program point p: the meaning of the pair (x, p) is that the assignment to x at point p is the current one. The initial value in this case is:

$$\mathrm{RD}_{init} = \{(x, ?) \mid x \text{ is a variable in the program}\}$$

Reaching Definitions is a forward analysis and we require the least (most precise) solutions to the set of equations.

Analysis Information

At each program point some definitions get "killed" (those which define the same variable as at the program point) while others are "generated".

A suitable representation for properties are sets of pairs, where each pair contains a variable x and a program point p: the meaning of the pair (x, p) is that the assignment to x at point p is the current one. The initial value in this case is:

$$\mathrm{RD}_{init} = \{(x, ?) \mid x \text{ is a variable in the program}\}$$

Reaching Definitions is a forward analysis and we require the least (most precise) solutions to the set of equations.

Analysis Information

At each program point some definitions get “killed” (those which define the same variable as at the program point) while others are “generated”.

A suitable representation for properties are sets of pairs, where each pair contains a variable x and a program point *p*: the meaning of the pair (x, p) is that the assignment to x at point *p* is the current one. The initial value in this case is:

$$\mathrm{RD}_{init} = \{(x, ?) \mid x \text{ is a variable in the program}\}$$

Reaching Definitions is a forward analysis and we require the least (most precise) solutions to the set of equations.

Analysis Information

At each program point some definitions get "killed" (those which define the same variable as at the program point) while others are "generated".

A suitable representation for properties are sets of pairs, where each pair contains a variable x and a program point p: the meaning of the pair (x, p) is that the assignment to x at point p is the current one. The initial value in this case is:

$$\mathrm{RD}_{init} = \{(x, ?) \mid x \text{ is a variable in the program}\}$$

Reaching Definitions is a forward analysis and we require the least (most precise) solutions to the set of equations.

Equations & Solutions

For our initial program fragment

```
$[m \leftarrow 2]^1$;
**while** $[n > 1]^2$ **do**
    $[m \leftarrow m \times n]^3$;
    $[n \leftarrow n - 1]^4$
**end while**
$[\textbf{stop}]^5$
```

some of the *RD* equations we get are:

$$\begin{aligned} \mathrm{RD}_{entry}(1) &= \{(m, ?), (n, ?)\} \\ \mathrm{RD}_{entry}(2) &= \mathrm{RD}_{exit}(1) \cup \mathrm{RD}_{exit}(4) \end{aligned}$$

Equations & Solutions

For our initial program fragment

```
$[m \leftarrow 2]^1$;
while $[n > 1]^2$ do
    $[m \leftarrow m \times n]^3$;
    $[n \leftarrow n - 1]^4$
end while
[stop]$^5$
```

some of the *RD* equations we get are:

$$\begin{aligned} \mathrm{RD}_{entry}(1) &= \{(m, ?), (n, ?)\} \\ \mathrm{RD}_{entry}(2) &= \mathrm{RD}_{exit}(1) \cup \mathrm{RD}_{exit}(4) \end{aligned}$$

Equations & Solutions

$$\begin{aligned} \mathsf{RD}_{entry}(1) &= \{(m,?),(n,?)\} \\ \mathsf{RD}_{entry}(2) &= \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(4) \end{aligned}$$

	RD_{entry}	RD_{exit}
1	$\{(m,?),(n,?)\}$	$\{(m,1),(n,?)\}$
2	$\{(m,1),(m,3),(n,?),(n,4)\}$	$\{(m,1),(m,3),(n,?),(n,4)\}$
3	$\{(m,1),(m,3),(n,?),(n,4)\}$	$\{(m,3),(n,?),(n,4)\}$
4	$\{(m,3),(n,?),(n,4)\}$	$\{(m,3),(n,4)\}$
5	$\{(m,1),(m,3),(n,?),(n,4)\}$	$\{(m,1),(m,3),(n,?),(n,4)\}$

Equations & Solutions

$$\begin{aligned}\mathrm{RD}_{entry}(1) &= \{(m, ?), (n, ?)\} \\ \mathrm{RD}_{entry}(2) &= \mathrm{RD}_{exit}(1) \cup \mathrm{RD}_{exit}(4)\end{aligned}$$

	RD_{entry}	RD_{exit}
1	$\{(m, ?), (n, ?)\}$	$\{(m, 1), (n, ?)\}$
2	$\{(m, 1), (m, 3), (n, ?), (n, 4)\}$	$\{(m, 1), (m, 3), (n, ?), (n, 4)\}$
3	$\{(m, 1), (m, 3), (n, ?), (n, 4)\}$	$\{(m, 3), (n, ?), (n, 4)\}$
4	$\{(m, 3), (n, ?), (n, 4)\}$	$\{(m, 3), (n, 4)\}$
5	$\{(m, 1), (m, 3), (n, ?), (n, 4)\}$	$\{(m, 1), (m, 3), (n, ?), (n, 4)\}$

Solving Equations

How can we construct solution to the data flow equations?
Answer: Iteratively, by improving approximations/guesses.

INPUT: Control Flow Graph
i.e. initial(p), pred(p).

OUTPUT: Reaching Definitions *RD*.

METHOD: Step 1: Initialisation
Step 2: Iteration

Solving Equations

How can we construct solution to the data flow equations?
Answer: Iteratively, by improving approximations/guesses.

INPUT: Control Flow Graph
i.e. initial(p), pred(p).

OUTPUT: Reaching Definitions *RD*.

METHOD: Step 1: Initialisation
Step 2: Iteration

Solving Equations

How can we construct solution to the data flow equations?
Answer: Iteratively, by improving approximations/guesses.

INPUT: Control Flow Graph
i.e. initial(p), pred(p).

OUTPUT: Reaching Definitions *RD*.

METHOD: Step 1: Initialisation
Step 2: Iteration

Solving Equations

How can we construct solution to the data flow equations?
Answer: Iteratively, by improving approximations/guesses.

INPUT: Control Flow Graph
i.e. initial(p), pred(p).

OUTPUT: Reaching Definitions *RD*.

METHOD: Step 1: Initialisation
Step 2: Iteration

Some Examples

Some examples of data flow analyses — and the possible applications and optimisations they allow for — are:

- Reaching Definitions — Constant Folding
- Available Expressions — Avoid Re-computations
- Very Busy Expressions — Hoisting
- Live Variables — Dead Code Elimination

- *Information Flow — Computer Security*
- *Shape Analyis — Pointer Analysis — etc.*

Some Examples

Some examples of data flow analyses — and the possible applications and optimisations they allow for — are:

- Reaching Definitions — Constant Folding
- Available Expressions — Avoid Re-computations
- Very Busy Expressions — Hoisting
- Live Variables — Dead Code Elimination

- *Information Flow — Computer Security*
- *Shape Analyis — Pointer Analysis — etc.*

Some Examples

Some examples of data flow analyses — and the possible applications and optimisations they allow for — are:

- Reaching Definitions — Constant Folding
- Available Expressions — Avoid Re-computations
- Very Busy Expressions — Hoisting
- Live Variables — Dead Code Elimination

- *Information Flow — Computer Security*
- *Shape Analyis — Pointer Analysis — etc.*

Some Examples

Some examples of data flow analyses — and the possible applications and optimisations they allow for — are:

- Reaching Definitions — Constant Folding
- Available Expressions — Avoid Re-computations
- Very Busy Expressions — Hoisting
- Live Variables — Dead Code Elimination

- *Information Flow — Computer Security*
- *Shape Analyis — Pointer Analysis — etc.*

Some Examples

Some examples of data flow analyses — and the possible applications and optimisations they allow for — are:

- Reaching Definitions — Constant Folding
- Available Expressions — Avoid Re-computations
- Very Busy Expressions — Hoisting
- Live Variables — Dead Code Elimination

- *Information Flow — Computer Security*
- *Shape Analyis — Pointer Analysis — etc.*

Some Examples

Some examples of data flow analyses — and the possible applications and optimisations they allow for — are:

- Reaching Definitions — Constant Folding
- Available Expressions — Avoid Re-computations
- Very Busy Expressions — Hoisting
- Live Variables — Dead Code Elimination

- *Information Flow — Computer Security*
- *Shape Analyis — Pointer Analysis — etc.*

Some Examples

Some examples of data flow analyses — and the possible applications and optimisations they allow for — are:

- Reaching Definitions — Constant Folding
- Available Expressions — Avoid Re-computations
- Very Busy Expressions — Hoisting
- Live Variables — Dead Code Elimination

- *Information Flow — Computer Security*
- *Shape Analyis — Pointer Analysis — etc.*

Code Optimisation

To illustrate the ideas we shall show how Reaching Definitions can be used to perform Constant Folding.

There are two ingredients to this:

- Replace the use of a variable in some expression by a constant if it is known that the value of that variable will always be a constant.
- Simplify an expression by partially evaluating it: subexpressions that contain no variables can be evaluated.

Code Optimisation

To illustrate the ideas we shall show how Reaching Definitions can be used to perform Constant Folding.

There are two ingredients to this:

- Replace the use of a variable in some expression by a constant if it is known that the value of that variable will always be a constant.
- Simplify an expression by partially evaluating it: subexpressions that contain no variables can be evaluated.

Code Optimisation

To illustrate the ideas we shall show how Reaching Definitions can be used to perform Constant Folding.

There are two ingredients to this:

- Replace the use of a variable in some expression by a constant if it is known that the value of that variable will always be a constant.
- Simplify an expression by partially evaluating it: subexpressions that contain no variables can be evaluated.

Code Optimisation

To illustrate the ideas we shall show how Reaching Definitions can be used to perform Constant Folding.

There are two ingredients to this:

- Replace the use of a variable in some expression by a constant if it is known that the value of that variable will always be a constant.
- Simplify an expression by partially evaluating it: subexpressions that contain no variables can be evaluated.

Constant Folding I

$$RD \vdash [\, x := a \,]^{\ell} \;\triangleright\; [\, x := a[y \mapsto n] \,]^{\ell}$$

$$\text{if} \begin{cases} y \in FV(a) \;\wedge\; (y, ?) \notin \mathrm{RD}_{entry}(\ell) \;\wedge \\ \forall (y', \ell') \in \mathrm{RD}_{entry}(\ell) : \\ y' = y \;\Rightarrow\; [\ldots]^{\ell'} = [\, y := n \,]^{\ell'} \end{cases}$$

$$RD \vdash [\, x := a \,]^{\ell} \;\triangleright\; [\, x := n \,]^{\ell}$$

$$\text{if} \begin{cases} FV(a) = \emptyset \;\wedge\; a \text{ is not constant} \;\wedge \\ a \text{ evaluates to } n \end{cases}$$

Constant Folding I

$$RD \vdash [\, x := a \,]^{\ell} \;\triangleright\; [\, x := a[y \mapsto n] \,]^{\ell}$$

$$\text{if} \begin{cases} y \in FV(a) \;\wedge\; (y, ?) \notin \mathrm{RD}_{entry}(\ell) \;\wedge \\ \forall (y', \ell') \in \mathrm{RD}_{entry}(\ell) : \\ y' = y \;\Rightarrow\; [\ldots]^{\ell'} = [\, y := n \,]^{\ell'} \end{cases}$$

$$RD \vdash [\, x := a \,]^{\ell} \;\triangleright\; [\, x := n \,]^{\ell}$$

$$\text{if} \begin{cases} FV(a) = \emptyset \;\wedge\; a \text{ is not constant} \;\wedge \\ a \text{ evaluates to } n \end{cases}$$

Constant Folding II

$$\frac{RD \vdash S_1 \;\rhd\; S_1'}{RD \vdash S_1 ; S_2 \;\rhd\; S_1' ; S_2}$$

$$\frac{RD \vdash S_2 \;\rhd\; S_2'}{RD \vdash S_1 ; S_2 \;\rhd\; S_1 ; S_2'}$$

$$\frac{RD \vdash S_1 \;\rhd\; S_1'}{RD \vdash \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2 \;\rhd\; \textbf{if } [b]^\ell \textbf{ then } S_1' \textbf{ else } S_2}$$

$$\frac{RD \vdash S_2 \;\rhd\; S_2'}{RD \vdash \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2 \;\rhd\; \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2'}$$

$$\frac{RD \vdash S \;\rhd\; S'}{RD \vdash \textbf{while } [b]^\ell \textbf{ do } S \;\rhd\; \textbf{while } [b]^\ell \textbf{ do } S'}$$

Constant Folding II

$$\frac{RD \vdash S_1 \;\triangleright\; S_1'}{RD \vdash S_1 ; S_2 \;\triangleright\; S_1' ; S_2}$$

$$\frac{RD \vdash S_2 \;\triangleright\; S_2'}{RD \vdash S_1 ; S_2 \;\triangleright\; S_1 ; S_2'}$$

$$\frac{RD \vdash S_1 \;\triangleright\; S_1'}{RD \vdash \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2 \;\triangleright\; \textbf{if } [b]^\ell \textbf{ then } S_1' \textbf{ else } S_2}$$

$$\frac{RD \vdash S_2 \;\triangleright\; S_2'}{RD \vdash \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2 \;\triangleright\; \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2'}$$

$$\frac{RD \vdash S \;\triangleright\; S'}{RD \vdash \textbf{while } [b]^\ell \textbf{ do } S \;\triangleright\; \textbf{while } [b]^\ell \textbf{ do } S'}$$

Constant Folding II

$$\frac{RD \vdash S_1 \;\triangleright\; S_1'}{RD \vdash S_1 ; S_2 \;\triangleright\; S_1' ; S_2}$$

$$\frac{RD \vdash S_2 \;\triangleright\; S_2'}{RD \vdash S_1 ; S_2 \;\triangleright\; S_1 ; S_2'}$$

$$\frac{RD \vdash S_1 \;\triangleright\; S_1'}{RD \vdash \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2 \;\triangleright\; \textbf{if } [b]^\ell \textbf{ then } S_1' \textbf{ else } S_2}$$

$$\frac{RD \vdash S_2 \;\triangleright\; S_2'}{RD \vdash \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2 \;\triangleright\; \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2'}$$

$$\frac{RD \vdash S \;\triangleright\; S'}{RD \vdash \textbf{while } [b]^\ell \textbf{ do } S \;\triangleright\; \textbf{while } [b]^\ell \textbf{ do } S'}$$

Constant Folding II

$$\frac{RD \vdash S_1 \;\triangleright\; S_1'}{RD \vdash S_1 ; S_2 \;\triangleright\; S_1' ; S_2}$$

$$\frac{RD \vdash S_2 \;\triangleright\; S_2'}{RD \vdash S_1 ; S_2 \;\triangleright\; S_1 ; S_2'}$$

$$\frac{RD \vdash S_1 \;\triangleright\; S_1'}{RD \vdash \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2 \;\triangleright\; \textbf{if } [b]^\ell \textbf{ then } S_1' \textbf{ else } S_2}$$

$$\frac{RD \vdash S_2 \;\triangleright\; S_2'}{RD \vdash \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2 \;\triangleright\; \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2'}$$

$$\frac{RD \vdash S \;\triangleright\; S'}{RD \vdash \textbf{while } [b]^\ell \textbf{ do } S \;\triangleright\; \textbf{while } [b]^\ell \textbf{ do } S'}$$

Constant Folding II

$$\frac{RD \vdash S_1 \ \triangleright\ S_1'}{RD \vdash S_1 ; S_2 \ \triangleright\ S_1' ; S_2}$$

$$\frac{RD \vdash S_2 \ \triangleright\ S_2'}{RD \vdash S_1 ; S_2 \ \triangleright\ S_1 ; S_2'}$$

$$\frac{RD \vdash S_1 \ \triangleright\ S_1'}{RD \vdash \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2 \ \triangleright\ \textbf{if } [b]^\ell \textbf{ then } S_1' \textbf{ else } S_2}$$

$$\frac{RD \vdash S_2 \ \triangleright\ S_2'}{RD \vdash \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2 \ \triangleright\ \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2'}$$

$$\frac{RD \vdash S \ \triangleright\ S'}{RD \vdash \textbf{while } [b]^\ell \textbf{ do } S \ \triangleright\ \textbf{while } [b]^\ell \textbf{ do } S'}$$

An Example

To illustrate the use of the transformation consider:

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

The (least) solution to the Reaching Definition analysis is:

$$\begin{aligned}
\mathrm{RD}_{entry}(1) &= \{(x,?),(y,?),(z,?)\}\\
\mathrm{RD}_{exit}(1) &= \{(x,1),(y,?),(z,?)\}\\[1ex]
\mathrm{RD}_{entry}(2) &= \{(x,1),(y,?),(z,?)\}\\
\mathrm{RD}_{exit}(2) &= \{(x,1),(y,2),(z,?)\}\\[1ex]
\mathrm{RD}_{entry}(3) &= \{(x,1),(y,2),(z,?)\}\\
\mathrm{RD}_{exit}(3) &= \{(x,1),(y,2),(z,3)\}
\end{aligned}$$

An Example

To illustrate the use of the transformation consider:

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

The (least) solution to the Reaching Definition analysis is:

$$\begin{aligned}
\mathrm{RD}_{entry}(1) &= \{(x, ?), (y, ?), (z, ?)\} \\
\mathrm{RD}_{exit}(1) &= \{(x, 1), (y, ?), (z, ?)\} \\[1ex]
\mathrm{RD}_{entry}(2) &= \{(x, 1), (y, ?), (z, ?)\} \\
\mathrm{RD}_{exit}(2) &= \{(x, 1), (y, 2), (z, ?)\} \\[1ex]
\mathrm{RD}_{entry}(3) &= \{(x, 1), (y, 2), (z, ?)\} \\
\mathrm{RD}_{exit}(3) &= \{(x, 1), (y, 2), (z, 3)\}
\end{aligned}$$

Constant Folding

We have for example the following:

$$RD \vdash [\, y := x + 10 \,]^2 \;\triangleright\; [\, y := 10 + 10 \,]^2$$

and therfore the rules for sequential composition allow us to do the following transformation:

$$\begin{aligned} RD \vdash\;\; & [\, x := 10 \,]^1; [\, y := x + 10 \,]^2; [\, z := y + 10 \,]^3 \;\triangleright \\ & [\, x := 10 \,]^1; [\, y := 10 + 10 \,]^2; [\, z := y + 10 \,]^3 \end{aligned}$$

Constant Folding

We have for example the following:

$$RD \vdash [\ y := x + 10\ ]^2 \ \triangleright\ [\ y := 10 + 10\ ]^2$$

and therfore the rules for sequential composition allow us to do the following transformation:

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

Transformation

We can continue this kind of transformation and obtain:

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

after which no more steps are possible.

Transformation

We can continue this kind of transformation and obtain:

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

after which no more steps are possible.

Transformation

We can continue this kind of transformation and obtain:

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

after which no more steps are possible.

Transformation

We can continue this kind of transformation and obtain:

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

after which no more steps are possible.

Transformation

We can continue this kind of transformation and obtain:

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

after which no more steps are possible.

Transformation

We can continue this kind of transformation and obtain:

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

after which no more steps are possible.

Transformation

We can continue this kind of transformation and obtain:

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

after which no more steps are possible.

Additional Issues

The above example shows that optimisation is in general the result of a number of successive transformations.

$$RD \vdash S_1 \;\triangleright\; S_2 \;\triangleright\; \ldots \;\triangleright\; S_n.$$

This could be costly because one S_1 has been transformed into S_2 we might have to *re-compute* the Reaching Definition analysis before the next transformation step can be done.

It could also be the case that different sequences of transformations either lead to different end results or are of very different length.

Additional Issues

The above example shows that optimisation is in general the result of a number of successive transformations.

$$RD \vdash S_1 \ \triangleright \ S_2 \ \triangleright \ \ldots \ \triangleright \ S_n.$$

This could be costly because one S_1 has been transformed into S_2 we might have to *re-compute* the Reaching Definition analysis before the next transformation step can be done.

It could also be the case that different sequences of transformations either lead to different end results or are of very different length.

Additional Issues

The above example shows that optimisation is in general the result of a number of successive transformations.

$$RD \vdash S_1 \ \triangleright\ S_2 \ \triangleright\ \ldots \ \triangleright\ S_n.$$

This could be costly because one S_1 has been transformed into S_2 we might have to *re-compute* the Reaching Definition analysis before the next transformation step can be done.

It could also be the case that different sequences of transformations either lead to different end results or are of very different length.

Lecture: Executive Summary

- Data Flow Analyis
- Monotone Frameworks
- Control Flow Analysis
- Abstract Interpretation
- Further Topics

Lecture: Executive Summary

- Data Flow Analyis
- Monotone Frameworks
- Control Flow Analysis
- Abstract Interpretation
- Further Topics

Lecture: Executive Summary

- Data Flow Analyis
- Monotone Frameworks
- Control Flow Analysis
- Abstract Interpretation
- Further Topics

Lecture: Executive Summary

- Data Flow Analyis
- Monotone Frameworks
- Control Flow Analysis
- Abstract Interpretation
- Further Topics

Lecture: Executive Summary

- Data Flow Analyis
- Monotone Frameworks
- Control Flow Analysis
- Abstract Interpretation
- Further Topics

Lecture: Milestones

Coursework I: Mid February – Test

Coursework II: Early March – Test

Examination: Sometime in Week 11

Lecture: Milestones

Coursework I: Mid February – Test

Coursework II: Early March – Test

Examination: Sometime in Week 11

Lecture: Milestones

Coursework I: Mid February – Test

Coursework II: Early March – Test

Examination: Sometime in Week 11