Program Analysis (70020)
Overview

Herbert Wiklicky

Department of Computing
Imperial College London

herbert@doc.ic.ac.uk
h.wiklicky@imperial.ac.uk

Autumn 2022
Lectures 11 October and 24 November 2022

Lecture Theatre 145 on Tuesday (2pm-4pm) and Thursday (9am-11am).
Lectures 11 October and 24 November 2022

Lecture Theatre 145 on Tuesday (2pm-4pm) and Thursday (9am-11am).

Tutorials typically Tuesdays, second hour.
Lectures 11 October and 24 November 2022

Lecture Theatre 145 on Tuesday (2pm-4pm) and Thursday (9am-11am).

Tutorials typically Tuesdays, second hour.

Coursework Tests on 27 October and 22 November
Lectures 11 October and 24 November 2022

Lecture Theatre 145 on Tuesday (2pm-4pm) and Thursday (9am-11am).

Tutorials typically Tuesdays, second hour.

Coursework Tests on 27 October and 22 November

Material and Notes on

https://www.doc.ic.ac.uk/~herbert/teaching.html

Scientia, Panopto (COMP97146/97128), etc.
Lectures 11 October and 24 November 2022

Lecture Theatre 145 on Tuesday (2pm-4pm)
    and Thursday (9am-11am).

Tutorials typically Tuesdays, second hour.

Coursework Tests on 27 October and 22 November

Material and Notes on

    https://www.doc.ic.ac.uk/~herbert/teaching.html
    Scientia, Panopto (COMP97146/97128), etc.

Assessment
Lectures 11 October and 24 November 2022

Lecture Theatre 145 on Tuesday (2pm-4pm) and Thursday (9am-11am).

Tutorials typically Tuesdays, second hour.

Coursework Tests on 27 October and 22 November

Material and Notes on

https://www.doc.ic.ac.uk/~herbert/teaching.html
Scientia, Panopto (COMP97146/97128), etc.

Assessment

Coursework Test I: Thu 27 October at 9:00
Lectures 11 October and 24 November 2022

Lecture Theatre 145 on Tuesday (2pm-4pm)
and Thursday (9am-11am).

Tutorials typically Tuesdays, second hour.

Coursework Tests on 27 October and 22 November

Material and Notes on

https://www.doc.ic.ac.uk/~herbert/teaching.html

Scientia, Panopto (COMP97146/97128), etc.

Assessment

Coursework Test I: Thu 27 October at 9:00
Coursework Test II: Tue 22 November at 14:00
Lectures 11 October and 24 November 2022

Lecture Theatre 145 on Tuesday (2pm-4pm) and Thursday (9am-11am).

Tutorials typically Tuesdays, second hour.

Coursework Tests on 27 October and 22 November

Material and Notes on

https://www.doc.ic.ac.uk/~herbert/teaching.html

Scientia, Panopto (COMP97146/97128), etc.

Assessment

Coursework Test I: Thu 27 October at 9:00

Coursework Test II: Tue 22 November at 14:00

Examination: Week 11, 12-16 December
Program analysis is an automated technique for finding out properties of programs without having to execute them.
Program analysis is an automated technique for finding out properties of programs without having to execute them.

Static Analysis vs Dynamic Testing
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 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
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

Unfortunately, the achieving the aims of (static) program analysis tend to be computationally extremely hard.
In some sense Program Analysis is an impossible task.
Program Properties

In some sense Program Analysis is an impossible task.

**Decidable Problems** There exists an algorithm or computational process which computes a solution (in finite time) for all instances of the problem.
In some sense Program Analysis is an impossible task.

**Decidable Problems** There exists an algorithm or computational process which computes a solution (in finite time) for all instances of the problem.

**Halting Problem** There is no general computational process or machine which can decide whether or not any given program terminates.
In some sense Program Analysis is an impossible task.

**Decidable Problems** There exists an algorithm or computational process which computes a solution (in finite time) for all instances of the problem.

**Halting Problem** There is no general computational process or machine which can decide whether or not any given program terminates.

**Rice Theorem** Any non-trivial program property is undecidable.
Program Properties

In some sense Program Analysis is an impossible task.

Decidable Problems There exists an algorithm or computational process which computes a solution (in finite time) for all instances of the problem.

Halting Problem There is no general computational process or machine which can decide whether or not any given program terminates.

Rice Theorem Any non-trivial program property is undecidable.

The approach is to find terminating algorithms for program analysis while not always finding a “meaningful” solution.
Fermat’s Program – Terminates

1: \( \text{try} \leftarrow \text{true}; \)
2: \( x \leftarrow 1; \)
3: \( \text{while try do} \)
4: \( y \leftarrow 1; \)
5: \( \text{while } y \leq x \text{ && try do} \)
6: \( z \leftarrow 1; \)
7: \( \text{while } z \leq y \text{ && try do} \)
8: \( \text{try} \leftarrow x^3 + y^3 \neq z^3 \)
9: \( z \leftarrow z + 1; \)
10: \( \text{end while} \)
11: \( y \leftarrow y + 1; \)
12: \( \text{end while} \)
13: \( x \leftarrow x + 1; \)
14: \( \text{end while} \)
Collatz Problem – Unknown

Take an integer $x$ and compute a sequence of updates:

1: while $x \neq 1$ do
2: if $x \mod 2 = 0$ then
3: $x \leftarrow x/2$;
4: else
5: $x \leftarrow 3 \times x + 1$
6: end if
7: end while

Currently it is unknown whether this terminates for all $x$. 
Collatz Problem – Unknown

Take an integer $x$ and compute a sequence of updates:

1: while $x \neq 1$ do
2:       if $x \mod 2 = 0$ then
3:           $x \leftarrow x/2$;
4:       else
5:           $x \leftarrow 3 \times x + 1$
6:       end if
7: end while

Currently it is unknown whether this terminates for all $x$. 
Some techniques used in program analysis include:

- Data Flow Analysis
- Control Flow Analysis
- Types and Effects Systems
- Abstract Interpretation
Some techniques used in program analysis include:

- Data Flow Analysis
Some techniques used in program analysis include:

- Data Flow Analysis
- Control Flow Analysis
Techniques

Some techniques used in program analysis include:

- Data Flow Analysis
- Control Flow Analysis
- Types and Effects Systems
Techniques

Some techniques used in program analysis include:

- Data Flow Analysis
- Control Flow Analysis
- Types and Effects Systems
- Abstract Interpretation
Some techniques used in program analysis include:

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

Some techniques used in program analysis include:

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


Techniques

Some techniques used in program analysis include:

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


A First Example

Consider the following fragment in *some* procedural language.

1: \( m \leftarrow 2; \)
2: \textbf{while} \( n > 1 \) \textbf{do}
3: \( m \leftarrow m \times n; \)
4: \( n \leftarrow n - 1 \)
5: \textbf{end while}
6: \textbf{stop}
A First Example

Consider the following fragment in *some* procedural language.

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

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

Consider the following fragment in *some* procedural language.

1: \( m \leftarrow 2; \)
2: while \( n > 1 \) do
3: \( m \leftarrow m \times n; \)
4: \( n \leftarrow n - 1 \)
5: end while
6: stop

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 \( m \), indepently of the initial values of \( m \) and \( n \).
A Parity Analysis

Claim: This program fragment always returns an even $m$, independently 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 Parity Analysis

**Claim:** This program fragment always returns an **even** $m$, independently 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
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
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
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 $n$.

1: $m \leftarrow 2$
2: while $n > 1$ do
3: $m \leftarrow m \times n$
4: $n \leftarrow n - 1$
5: end while
6: stop
A First Example

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

1: \( m \leftarrow 2; \)  
2: while \( n > 1 \) do  
3: \( m \leftarrow m \times n; \)  
4: \( n \leftarrow n - 1 \)  
5: end while  
6: stop
A First Example

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

1: \( m \leftarrow 2; \) \[\triangleright \text{unknown}(m) - \text{unknown}(n)\]
2: \( \textbf{while} \ n > 1 \ \textbf{do} \)
3: \( m \leftarrow m \times n; \) \[\triangleright \text{even}(m) - \text{unknown}(n)\]
4: \( n \leftarrow n - 1 \)
5: \( \textbf{end while} \)
6: \( \textbf{stop} \)
A First Example

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

1: \( m \leftarrow 2; \) \hspace{1cm} \triangleright \text{unknown}(m) – \text{unknown}(n)
2: \textbf{while} \( n > 1 \) \textbf{do} \hspace{1cm} \triangleright \text{even}(m) – \text{unknown}(n)
3: \hspace{1cm} m \leftarrow m \times n; \hspace{1cm} \triangleright \text{even}(m) – \text{unknown}(n)
4: \hspace{1cm} n \leftarrow n – 1 \hspace{1cm} \triangleright \text{even}(m) – \text{unknown}(n)
5: \textbf{end while} \hspace{1cm} \triangleright \text{even}(m) – \text{unknown}(n)
6: \textbf{stop}
A First Example

Executing the program with \emph{abstract} values, parity, for \textit{m} and \textit{n}.

1: \textit{m} ← 2;  \quad \triangleright \ \text{unknown(m) – unknown(n)}
2: \textbf{while} \textit{n} > 1 \textbf{do}  \quad \triangleright \ \text{even(m) – unknown(n)}
3: \quad \textit{m} ← \textit{m} × \textit{n};  \quad \triangleright \ \text{even(m) – unknown(n)}
4: \quad \textit{n} ← \textit{n} – 1  \quad \triangleright \ \text{even(m) – unknown(n)}
5: \quad \textbf{end while}  \quad \triangleright \ \text{even(m) – unknown(n)}
6: \textbf{stop}  \quad \triangleright \ \text{even(m) – unknown(n)}
A First Example

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

1: \( m \leftarrow 2; \) ▷ unknown(m) – unknown(n)
2: \textbf{while} \( n > 1 \) \textbf{do} ▷ even(m) – unknown(n)
3: \hspace{1em} \( m \leftarrow m \times n; \) ▷ even(m) – unknown(n)
4: \hspace{1em} \( n \leftarrow n - 1 \) ▷ even(m) – unknown(n)
5: \textbf{end while} ▷ even(m) – unknown(n)
6: \textbf{stop} ▷ even(m) – unknown(n)
A First Example

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

1: \( m \leftarrow 2; \) ▷ unknown(m) – unknown(n)
2: while \( n > 1 \) do ▷ even(m) – unknown(n)
3: \( m \leftarrow m \times n; \)
4: \( n \leftarrow n - 1 \)
5: end while
6: stop ▷ even(m) – unknown(n)
A First Example

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

1: \( m \leftarrow 2; \)  \( \triangleright \) unknown(\( m \)) – unknown(\( n \))
2: \( \textbf{while} \ n > 1 \ \textbf{do} \)  \( \triangleright \) even(\( m \)) – unknown(\( n \))
3: \( \quad m \leftarrow m \times n; \)  \( \triangleright \) even(\( m \)) – unknown(\( n \))
4: \( \quad n \leftarrow n - 1 \)  \( \triangleright \) even(\( m \)) – unknown(\( n \))
5: \( \textbf{end while} \)  \( \triangleright \) even(\( m \)) – unknown(\( n \))
6: \( \textbf{stop} \)
A First Example

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

1: \[ m \leftarrow 2; \quad \triangleright \text{unknown}(m) - \text{unknown}(n) \]
2: \[ \textbf{while } n > 1 \textbf{ do} \quad \triangleright \text{even}(m) - \text{unknown}(n) \]
3: \[ m \leftarrow m \times n; \quad \triangleright \text{even}(m) - \text{unknown}(n) \]
4: \[ n \leftarrow n - 1 \quad \triangleright \text{even}(m) - \text{unknown}(n) \]
5: \[ \textbf{end while} \]
6: \[ \textbf{stop} \quad \triangleright \text{even}(m) - \text{unknown}(n) \]
A First Example

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

1: $m \leftarrow 2$;  \hspace{1cm} \triangleright \text{unknown}(m) – \text{unknown}(n)
2: \textbf{while} $n > 1$ \textbf{do}  \hspace{1cm} \triangleright \text{even}(m) – \text{unknown}(n)
3: \hspace{1cm} m \leftarrow m \times n;  \hspace{1cm} \triangleright \text{even}(m) – \text{unknown}(n)
4: \hspace{1cm} n \leftarrow n – 1  \hspace{1cm} \triangleright \text{even}(m) – \text{unknown}(n)
5: \textbf{end while}  \hspace{1cm} \triangleright \text{even}(m) – \text{unknown}(n)
6: \textbf{stop}  \hspace{1cm} \triangleright \text{even}(m) – \text{unknown}(n)
A First Example

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

1: \( m \leftarrow 2; \)
2: while \( n > 1 \) do
3: \( m \leftarrow m \times n; \)
4: \( n \leftarrow n – 1 \)
5: end while
6: stop
A First Example

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

1: \( m \leftarrow 2; \)  \( \triangleright \) unknown(\( m \)) – unknown(\( n \))
2: \textbf{while } \( n > 1 \) \textbf{do}  \( \triangleright \) even(\( m \)) – unknown(\( n \))
3: \( m \leftarrow m \times n; \)  \( \triangleright \) even(\( m \)) – unknown(\( n \))
4: \( n \leftarrow n - 1 \)  \( \triangleright \) even(\( m \)) – unknown(\( n \))
5: \textbf{end while}  \( \triangleright \) even(\( m \)) – unknown(\( n \))
6: \textbf{stop}  \( \triangleright \) even(\( m \)) – unknown(\( n \))

Important: We can restart the loop!
The first program computes 2 times the factorial for any positive value of \( n \). Replacing ‘2’ by ‘1’ in the first statement gives:

1. \( m \leftarrow 1; \)
2. \textbf{while} \( n > 1 \) \textbf{do}
3. \hspace{1em} \( m \leftarrow m \times n; \)
4. \hspace{1em} \( n \leftarrow n – 1 \)
5. \textbf{end while}
6. \textbf{stop}
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:

1: $m \leftarrow 1$; $
2: \text{while } n > 1 \text{ do}$
3: $m \leftarrow m \times n$;
4: $n \leftarrow n - 1$
5: end while
6: stop

\[\triangleright\text{unknown}(m) - \text{unknown}(n)\]
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:

1: \( m \leftarrow 1 \); \( \triangleright \) unknown(m) – unknown(n)
2: while \( n > 1 \) do
3: \( m \leftarrow m \times n \); \( \triangleright \) odd(m) – unknown(n)
4: \( n \leftarrow n - 1 \)
5: end while
6: stop
The first program computes 2 times the factorial for any positive value of $n$. Replacing ‘2’ by ‘1’ in the first statement gives:

1: $m \leftarrow 1$
2: while $n > 1$ do
3: \hspace{1em} $m \leftarrow m \times n$
4: \hspace{1em} $n \leftarrow n - 1$
5: end while
6: stop

▷ odd(m) – unknown(n)
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:

1: \( m \leftarrow 1; \)  \( \triangleright \) unknown(m) – unknown(n)
2: \textbf{while} \( n > 1 \) \textbf{do}  \( \triangleright \) odd(m) – unknown(n)
3: \( m \leftarrow m \times n; \)  \( \triangleright \) odd(m) – unknown(n)
4: \( n \leftarrow n - 1 \)  \( \triangleright \) unknown(m) – unknown(n)
5: \textbf{end while}  \( \triangleright \) odd(m) – unknown(n)
6: \textbf{stop}
The first program computes 2 times the factorial for any positive value of \( n \). Replacing ‘2’ by ‘1’ in the first statement gives:

1: \( m \leftarrow 1 \); 
2: while \( n > 1 \) do  
3: \( m \leftarrow m \times n \); 
4: \( n \leftarrow n - 1 \); 
5: end while 
6: stop
The first program computes 2 times the factorial for any positive value of $n$. Replacing ‘2’ by ‘1’ in the first statement gives:

1: $m \leftarrow 1$;  
2: while $n > 1$ do  
3: $m \leftarrow m \times n$;  
4: $n \leftarrow n - 1$  
5: end while  
6: stop  

▷ unknown($m$) – unknown($n$)
▷ unknown($m$) – unknown($n$)
▷ odd($m$) – unknown($n$)
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:

1: \( m \leftarrow 1; \)
2: \( \text{while } n > 1 \text{ do} \)
3: \( m \leftarrow m \times n; \)
4: \( n \leftarrow n - 1 \)
5: \( \text{end while} \)
6: \( \text{stop} \)

\( \triangleright \text{unknown}(m) - \text{unknown}(n) \)

\( \triangleright \text{unknown}(m) - \text{unknown}(n) \)

\( \triangleright \text{unknown}(m) - \text{unknown}(n) \)

\( \triangleright \text{unknown}(m) - \text{unknown}(n) \)

\( \triangleright \text{unknown}(m) - \text{unknown}(n) \)
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:

1: \( m \leftarrow 1; \)  \hspace{1em} \triangleright \text{unknown}(m) - \text{unknown}(n)
2: \textbf{while } n > 1 \textbf{ do}  \hspace{1em} \triangleright \text{unknown}(m) - \text{unknown}(n)
3: \hspace{1em} m \leftarrow m \times n;  \hspace{1em} \triangleright \text{unknown}(m) - \text{unknown}(n)
4: \hspace{1em} n \leftarrow n - 1  \hspace{1em} \triangleright \text{unknown}(m) - \text{unknown}(n)
5: \hspace{1em} \textbf{end while}  \hspace{1em} \triangleright \text{unknown}(m) - \text{unknown}(n)
6: \hspace{1em} \textbf{stop}  \hspace{1em} \triangleright \text{unknown}(m) - \text{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
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 \( n \leq 1 \).
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.
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.
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*,

▶ yes means *definitely yes*,
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.
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.
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.
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

\[ z := 1 \]

\[ x > 0 \]

\[ z := z \times y \]

\[ x := x - 1 \]

This allows us to determine the predecessors \( \text{pred} \) and successors \( \text{succ} \) of each statement, e.g. \( \text{pred}(2) = \{1, 4\} \).
This allows us to determine the predecessors \textit{pred} and successors \textit{succ} of each statement, e.g. \textit{pred}(2) = \{1, 4\}.
Reaching Definition (RD) analysis determines which set of definitions (i.e. assignments) are current when control reaches a certain program point $p$. 
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:
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:

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

\[
\begin{align*}
\text{RD}_{\text{entry}}(p) &= \begin{cases} \\
\text{RD}_{\text{init}} & \text{if } p \text{ is initial} \\
\bigcup_{p' \in \text{pred}(p)} \text{RD}_{\text{exit}}(p') & \text{otherwise}
\end{cases} \\
\text{RD}_{\text{exit}}(p) &= (\text{RD}_{\text{entry}}(p) \setminus \text{kill}_{\text{RD}}(p)) \cup \text{gen}_{\text{RD}}(p)
\end{align*}
\]
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: \( RD_{init} = \{ (x, ?) | x \text{ is a variable in the program} \} \)
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.
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:

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

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

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

\[
\begin{align*}
&m \leftarrow 2^1; \\
\textbf{while } [n > 1]^2 \textbf{ do } \\
&m \leftarrow m \times n^3; \\
&[n \leftarrow n - 1]^4 \\
\textbf{end while } \\
&[\text{stop}]^5
\end{align*}
\]
Equations & Solutions

For our initial program fragment

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

some of the \textit{RD} equations we get are:

\[
\begin{align*}
\text{RD}_{\text{entry}}(1) &= \{(m,?), (n,?)\} \\
\text{RD}_{\text{entry}}(2) &= \text{RD}_{\text{exit}}(1) \cup \text{RD}_{\text{exit}}(4)
\end{align*}
\]
Equations & Solutions

\[
RD_{entry}(1) = \{(m, ?), (n, ?)\}
\]
\[
RD_{entry}(2) = RD_{exit}(1) \cup RD_{exit}(4)
\]
Equations & Solutions

\[ RD_{entry}(1) = \{(m, ?), (n, ?)\} \]
\[ RD_{entry}(2) = RD_{exit}(1) \cup RD_{exit}(4) \]

<table>
<thead>
<tr>
<th>RD_{entry}</th>
<th>RD_{exit}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 {(m, ?), (n, ?)}</td>
<td>2 {(m, 1), (m, 3), (n, ?), (n, 4)}</td>
</tr>
<tr>
<td>2 {(m, 1), (m, 3), (n, ?), (n, 4)}</td>
<td>3 {(m, 3), (n, ?), (n, 4)}</td>
</tr>
<tr>
<td>3 {(m, 1), (m, 3), (n, ?), (n, 4)}</td>
<td>4 {(m, 3), (n, 4)}</td>
</tr>
<tr>
<td>4 {(m, 3), (n, ?), (n, 4)}</td>
<td>5 {(m, 1), (m, 3), (n, ?), (n, 4)}</td>
</tr>
<tr>
<td>5 {(m, 1), (m, 3), (n, ?), (n, 4)}</td>
<td></td>
</tr>
</tbody>
</table>
Solving Equations

How can we construct solution to the data flow equations? Answer: Iteratively, by improving approximations/guesses.
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).
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.
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
- (Probabilistic) Program Slicing
- Shape Analysis — 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
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
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
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
- (Probabilistic) Program Slicing
- Shape Analysis — 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*
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
- (Probabilistic) Program Slicing
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
- (Probabilistic) Program Slicing
- Shape Analysis — Pointer Analysis — etc.
To illustrate the ideas we shall show how Reaching Definitions can be used to perform Constant Folding.
To illustrate the ideas we shall show how Reaching Definitions can be used to perform Constant Folding.

There are two ingredients to this:
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.
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

\[ RD \vdash [ x := a ]^\ell \triangleright [ x := a[y \mapsto n] ]^\ell \]

if \[
\begin{cases}
y \in FV(a) \land (y, ?) \notin RD_{entry}(\ell) \land \\
\forall (y', \ell') \in RD_{entry}(\ell) : \\
y' = y \Rightarrow [\ldots]^\ell' = [ y := n ]^\ell'
\end{cases}
\]
Constant Folding I

\[
RD \vdash [x := a]^{\ell} \triangleright [x := a[y \leftarrow n]]^{\ell}
\]

if \begin{align*}
y \in \text{FV}(a) & \land (y, ?) \notin \text{RD}_{\text{entry}}(\ell) \land \\
\forall (y', \ell') & \in \text{RD}_{\text{entry}}(\ell) : \\
y' = y & \Rightarrow \ldots = [y := n]^{\ell'}
\end{align*}

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

if \begin{align*}
\text{FV}(a) = \emptyset & \land a \text{ is not constant} \land \\
a \text{ evaluates to } n
\end{align*}
Constant Folding II

\[
\begin{align*}
RD & \vdash S_1 \triangleright S'_1 \\
\hline
RD & \vdash S_1; S_2 \triangleright S'_1; S_2
\end{align*}
\]
Constant Folding II

\[ RD \vdash S_1 \triangleright S'_1 \]
\[ RD \vdash S_1; S_2 \triangleright S'_1; S_2 \]
\[ RD \vdash S_2 \triangleright S'_2 \]
\[ RD \vdash S_1; S_2 \triangleright S_1; S'_2 \]
Constant Folding II

\[
\begin{align*}
RD & \vdash S_1 \triangleright S'_1 \\
RD & \vdash S_1 ; S_2 \triangleright S'_1 ; S_2 \\
RD & \vdash S_2 \triangleright S'_2 \\
RD & \vdash S_1 ; S_2 \triangleright S_1 ; S'_2 \\
RD & \vdash S_1 \triangleright S'_1 \\
RD & \vdash \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \triangleright \text{if } [b]^\ell \text{ then } S'_1 \text{ else } S_2
\end{align*}
\]
Constant Folding II

\[
\begin{align*}
RD \vdash S_1 & \triangleright S'_1 \\
\implies RD \vdash S_1; S_2 & \triangleright S'_1; S_2 \\
RD \vdash S_2 & \triangleright S'_2 \\
\implies RD \vdash S_1; S_2 & \triangleright S_1; S'_2 \\
RD \vdash S_1 & \triangleright S'_1 \\
\implies RD \vdash \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 & \triangleright \text{if } [b]^\ell \text{ then } S'_1 \text{ else } S_2 \\
RD \vdash S_2 & \triangleright S'_2 \\
\implies RD \vdash \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 & \triangleright \text{if } [b]^\ell \text{ then } S_1 \text{ else } S'_2
\end{align*}
\]
Constant Folding II

\[
\begin{align*}
RD &
\vdash \ S_1 \triangleright \ S'_1 \\
RD &
\vdash \ S_1; \ S_2 \triangleright \ S'_1; \ S_2 \\
RD &
\vdash \ S_2 \triangleright \ S'_2 \\
RD &
\vdash \ S_1; \ S_2 \triangleright \ S_1; \ S'_2 \\
RD &
\vdash \ S_1 \triangleright \ S'_1 \\
RD &
\vdash \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \triangleright \text{if } [b]^\ell \text{ then } S'_1 \text{ else } S_2 \\
RD &
\vdash \ S_2 \triangleright \ S'_2 \\
RD &
\vdash \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \triangleright \text{if } [b]^\ell \text{ then } S_1 \text{ else } S'_2 \\
RD &
\vdash \ S \triangleright \ S' \\
RD &
\vdash \text{while } [b]^\ell \text{ do } S \triangleright \text{while } [b]^\ell \text{ do } S'
\end{align*}
\]
An Example

To illustrate the use of the transformation consider:

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

To illustrate the use of the transformation consider:

\[
[ \ x := 10 \ ]; \ [ \ y := x + 10 \ ]; \ [ \ z := y + 10 \ ]
\]

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

\[
\text{RD}_{\text{entry}}(1) = \{(x,?), (y,?), (z,?)\}
\]
\[
\text{RD}_{\text{exit}}(1) = \{(x,1), (y,?), (z,?)\}
\]
\[
\text{RD}_{\text{entry}}(2) = \{(x,1), (y,?), (z,?)\}
\]
\[
\text{RD}_{\text{exit}}(2) = \{(x,1), (y,2), (z,?)\}
\]
\[
\text{RD}_{\text{entry}}(3) = \{(x,1), (y,2), (z,?)\}
\]
\[
\text{RD}_{\text{exit}}(3) = \{(x,1), (y,2), (z,3)\}
\]
We have for example the following:

\[ RD \vdash [y := x + 10]^2 \succ [y := 10 + 10]^2 \]
Constant Folding

We have for example the following:

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

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

\[
RD \vdash [ x := 10 ]^1; [ y := x + 10 ]^2; [ z := y + 10 ]^3 \triangleright [ x := 10 ]^1; [ y := 10 + 10 ]^2; [ z := y + 10 ]^3
\]
We can continue this kind of transformation and obtain:
Transformation

We can continue this kind of transformation and obtain:

\[ RD \vdash [ x := 10 ]^1; [ y := x + 10 ]^2; [ z := y + 10 ]^3 \]
We can continue this kind of transformation and obtain:

\[
\begin{align*}
RD \quad &\vdash \quad [x := 10]^1; [y := x + 10]^2; [z := y + 10]^3 \\
\triangleright &\quad [x := 10]^1; [y := 10 + 10]^2; [z := y + 10]^3
\end{align*}
\]
Transformation

We can continue this kind of transformation and obtain:

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

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

\[ \Rightarrow [ x := 10 ]^1; [ y := 20 ]^2; [ z := y + 10 ]^3 \]
Transformation

We can continue this kind of transformation and obtain:

\[
\begin{align*}
RD & \vdash [x := 10]_1; [y := x + 10]_2; [z := y + 10]_3 \\
\triangleright & [x := 10]_1; [y := 10 + 10]_2; [z := y + 10]_3 \\
\triangleright & [x := 10]_1; [y := 20]_2; [z := y + 10]_3 \\
\triangleright & [x := 10]_1; [y := 20]_2; [z := 20 + 10]_3
\end{align*}
\]
We can continue this kind of transformation and obtain:

\[
\begin{align*}
RD & \vdash [x := 10]_1; [y := x + 10]_2; [z := y + 10]_3 \\
& \quad \triangleright [x := 10]_1; [y := 10 + 10]_2; [z := y + 10]_3 \\
& \quad \triangleright [x := 10]_1; [y := 20]_2; [z := y + 10]_3 \\
& \quad \triangleright [x := 10]_1; [y := 20]_2; [z := 20 + 10]_3 \\
& \quad \triangleright [x := 10]_1; [y := 20]_2; [z := 30]_3
\end{align*}
\]
We can continue this kind of transformation and obtain:

\[
\text{RD} \vdash [x := 10]^1; [y := x + 10]^2; [z := y + 10]^3
\]

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

\[\uparrow\]
\[
[x := 10]^1; [y := 20]^2; [z := y + 10]^3
\]

\[\uparrow\]
\[
[x := 10]^1; [y := 20]^2; [z := 20 + 10]^3
\]

\[\uparrow\]
\[
[x := 10]^1; [y := 20]^2; [z := 30]^3
\]

after which no more steps are possible.
The above example shows that optimisation is in general the result of a number of successive transformations.

\[ RD \vdash S_1 \triangleright S_2 \triangleright \ldots \triangleright S_n. \]
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.
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 once \( S_1 \) has been transformed into \( S_2 \), we might have to \textit{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.
Correctness

Any Program Analysis should be:

▶ unambiguously specified
▶ efficiently computable
▶ most importantly: correct.
Correctness

Any Program Analysis should be:

- unambiguously specified,
Correctness

Any Program Analysis should be:
- unambiguously specified,
- efficiently computable,
Correctness

Any Program Analysis should be:

- unambiguously specified,
- efficiently computable,
- most importantly: correct.

For example, why not consider in RD before:

\[
\text{RD}^{\text{entry}}(2) = \text{RD}^{\text{exit}}(1) \cap \text{RD}^{\text{exit}}(4)
\]

instead of:

\[
\text{RD}^{\text{entry}}(2) = \text{RD}^{\text{exit}}(1) \cup \text{RD}^{\text{exit}}(4)
\]

It requires formal (mathematical) proof whether an analysis (or program transformation) is correct with respect to some model of execution or semantics.
Correctness

Any Program Analysis should be:

- unambiguously specified,
- efficiently computable,
- most importantly: correct.

For example, why not consider in $RD$ before:

$$RD_{entry}(2) = \bigcap RD_{exit}(1) \cap RD_{exit}(4)$$

instead of $RD_{entry}(2) = RD_{exit}(1) \cup RD_{exit}(4)$. 
Correctness

Any Program Analysis should be:

- unambiguously specified,
- efficiently computable,
- most importantly: correct.

For example, why not consider in $RD$ before:

$$RD_{entry}(2) = RD_{exit}(1) \cap RD_{exit}(4)$$

instead of $RD_{entry}(2) = RD_{exit}(1) \cup RD_{exit}(4)$.

It requires formal (mathematical) proof whether an analysis (or program transformation) is correct with respect to some model of execution or semantics.
A **program** is foremost a text but it has intended **meaning** or **semantics** describing its execution.
A program is foremost a text but it has intended meaning or semantics describing its execution.

A simple example: Why is $0.\dot{9} = 0.99999\ldots = 1$?
Formal Semantics

A **program** is foremost a text but it has intended *meaning* or *semantics* describing its execution.

A simple example: Why is \(0.\dot{9} = 0.99999 \ldots = 1?\)

Obviously, these are different strings! However, they have a meaning or semantics as specification of a real number in \(\mathbb{R}\).
A program is foremost a text but it has intended meaning or semantics describing its execution.

A simple example: Why is $0.\dot{9} = 0.99999\ldots = 1$?

Obviously, these are different strings! However, they have a meaning or semantics as specification of a real number in $\mathbb{R}$. More concretely, infinite strings refer to the limit of their expansion, so $[0.\dot{9}] = \lim(0.9, 0.99, 0.999, \ldots) = 1 = [1]$. 

A program is foremost a text but it has intended meaning or semantics describing its execution.

A simple example: Why is \(0.\dot{9} = 0.99999\ldots = 1\)?

Obviously, these are different strings! However, they have a meaning or semantics as specification of a real number in \(\mathbb{R}\). More concretely, infinite strings refer to the limit of their expansion, so \([0.\dot{9}] = \lim(0.9, 0.99, 0.999, \ldots) = 1 = [1]\).

This course will mostly be concerned with intuitive or light-weight semantics when it comes to the “meaning” of a program and the correctness of a program analysis.
Topics Covered – Executive Summary

- Data Flow Analysis
Topics Covered – Executive Summary

- Data Flow Analysis
- Monotone Frameworks
Topics Covered – Executive Summary

- Data Flow Analysis
- Monotone Frameworks
- Control Flow Analysis
Topics Covered – Executive Summary

- Data Flow Analysis
- Monotone Frameworks
- Control Flow Analysis
- Abstract Interpretation
Topics Covered – Executive Summary

- Data Flow Analysis
- Monotone Frameworks
- Control Flow Analysis
- Abstract Interpretation
- Probabilistic Programs
Topics Covered – Executive Summary

- Data Flow Analysis
- Monotone Frameworks
- Control Flow Analysis
- Abstract Interpretation
- Probabilistic Programs
- Probabilistic Abstract Interpretation
Topics Covered – Executive Summary

- Data Flow Analysis
- Monotone Frameworks
- Control Flow Analysis
- Abstract Interpretation
- Probabilistic Programs
- Probabilistic Abstract Interpretation
- Further Topics