Program Analysis (CO470/97128/97146)

Overview

Herbert Wiklicky

Department of Computing
Imperial College London

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

Autumn 2020
Lectures between 9 October and 27 November 2020

Lectures on Monday (10am-12noon)
and Friday (10am-12noon).
Lectures between 9 October and 27 November 2020

Lectures on Monday (10am-12noon) and Friday (10am-12noon).

Tutorials to be decided.
Lectures between 9 October and 27 November 2020

Lectures on Monday (10am-12noon) and Friday (10am-12noon).

Tutorials to be decided.

Coursework Tests to be decided.
Lectures between 9 October and 27 November 2020

Lectures on Monday (10am-12noon) and Friday (10am-12noon).

Tutorials to be decided.

Coursework Tests to be decided.

Material and Notes (CATE link) on

https://www.doc.ic.ac.uk/~herbert/teaching.html
Lectures between 9 October and 27 November 2020

Lectures on Monday (10am-12noon) and Friday (10am-12noon).

Tutorials to be decided.

Coursework Tests to be decided.

Material and Notes (CATE link) on

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

Assessment
Lectures between 9 October and 27 November 2020

Lectures on Monday (10am-12noon)  
and Friday (10am-12noon).

Tutorials to be decided.

Coursework Tests to be decided.

Material and Notes (CATE link) on

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

Assessment

Coursework Test I: End October
Lectures between 9 October and 27 November 2020

Lectures on Monday (10am-12noon) and Friday (10am-12noon).

Tutorials to be decided.

Coursework Tests to be decided.

Material and Notes (CATE link) on

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

Assessment

Coursework Test I:  End October
Coursework Test II:  Mid November
Lectures between 9 October and 27 November 2020

Lectures on Monday (10am-12noon) and Friday (10am-12noon).

Tutorials to be decided.

Coursework Tests to be decided.

Material and Notes (CATE link) on

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

Assessment

Coursework Test I: End October
Coursework Test II: Mid November

Examination: Week 10 or 11, i.e. 7/12 – 18/12
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

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

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
Some techniques used in program analysis include:

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

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*
A First Example

Consider the following fragment in some procedural language.

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

\[ m \leftarrow 2 \]
\[ \textbf{while} \ [n > 1] \]
\[ m \leftarrow m \times n \]
\[ n \leftarrow n - 1 \]
\textbf{end while}
\[ \textbf{stop} \]

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.

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}

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 \( m \), independently 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 from the start of the program.
Properties

We will assign to each variable one of three properties:
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: \hspace{1em} $m \leftarrow m \times n$;
4: \hspace{1em} $n \leftarrow n - 1$
5: end while
6: stop
A First Example

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

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

Executing the program with \textit{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{0.5cm} m \leftarrow m \times n;\)
4: \hspace{0.5cm} n \leftarrow n - 1
5: \hspace{0.5cm} \textbf{end while}
6: \textbf{stop}
A First Example

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

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

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

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

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(\( n \))
A First Example

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

1: \texttt{m ← 2}; \quad \triangleright \text{unknown}(m) – \text{unknown}(n)
2: \textbf{while } \texttt{n > 1} \textbf{ do} \quad \triangleright \text{even}(m) – \text{unknown}(n)
3: \quad \texttt{m ← m × n;} \quad \triangleright \text{even}(m) – \text{unknown}(n)
4: \quad \texttt{n ← n − 1} \quad \triangleright \text{even}(m) – \text{unknown}(n)
5: \quad \textbf{end while}
6: \texttt{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 \textit{abstract} values, parity, for \texttt{m} and \texttt{n}.

1: \texttt{m} ← 2;
2: \texttt{while} \texttt{n} > 1 \texttt{do}
3: \texttt{m} ← \texttt{m} × \texttt{n};
4: \texttt{n} ← \texttt{n} − 1
5: \texttt{end while}
6: \texttt{stop}

▷ \texttt{unknown(m)} − \texttt{unknown(n)}
▷ \texttt{even(m)} − \texttt{unknown(n)}
▷ \texttt{even(m)} − \texttt{unknown(n)}
▷ \texttt{even(m)} − \texttt{unknown(n)}
▷ \texttt{even(m)} − \texttt{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: \( m \leftarrow m \times n; \)     \( \triangleright \) 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)
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

▷ unknown($m$) – unknown($n$)
▷ even($m$) – unknown($n$)
▷ even($m$) – unknown($n$)
▷ even($m$) – unknown($n$)
▷ even($m$) – unknown($n$)
▷ 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: while \( n > 1 \) 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: end while  \( \triangleright \) even(m) – unknown(n)
6: stop  \( \triangleright \) 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:

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: \( m \leftarrow 1; \)    \( \triangleright \) unknown(m) – unknown(n)
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: \( m \leftarrow 1 \);  \hspace{1cm} \triangleright \text{unknown}(m) - \text{unknown}(n)
2: \textbf{while } n > 1 \textbf{ do}  \hspace{1cm} \triangleright \text{odd}(m) - \text{unknown}(n)
3: \hspace{1.5cm} m \leftarrow m \times n;  \hspace{1cm} \triangleright \text{unknown}(m) - \text{unknown}(n)
4: \hspace{1.5cm} 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: $m \leftarrow 1$; ▷ unknown($m$) – unknown($n$)
2: while $n > 1$ do ▷ odd($m$) – unknown($n$)
3: $m \leftarrow m \times n$; ▷ odd($m$) – unknown($n$)
4: $n \leftarrow n - 1$ ▷ odd($m$) – unknown($n$)
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: \[ m \leftarrow 1; \]  
2: \[ \textbf{while } n > 1 \textbf{ do} \]  
3: \[ m \leftarrow m \times n; \]  
4: \[ n \leftarrow n - 1 \]  
5: \[ \textbf{end while} \]  
6: \[ \textbf{stop} \]  

\[ \triangleright \text{unknown}(m) - \text{unknown}(n) \]  
\[ \triangleright \text{odd}(m) - \text{unknown}(n) \]  
\[ \triangleright \text{odd}(m) - \text{unknown}(n) \]  
\[ \triangleright \text{unknown}(m) - \text{unknown}(n) \]  
\[ \triangleright \text{odd}(m) - \text{unknown}(n) \]
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:  $m \leftarrow 1$
2:  while $n > 1$ do
3:     $m \leftarrow m \times n$
4:      $n \leftarrow n - 1$
5:  end while
6:  stop

$\triangleright$ unknown($m$) – unknown($n$)  
$\triangleright$ 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:

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

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

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

\( \triangleright \text{odd}(m) - \text{unknown}(n) \)
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: $m ← 1$;  
2: while $n > 1$ do  
3: $m ← m \times n$;  
4: $n ← n − 1$  
5: end while  
6: stop

△ $\text{unknown}(m) − \text{unknown}(n)$  
△ $\text{unknown}(m) − \text{unknown}(n)$  
△ $\text{unknown}(m) − \text{unknown}(n)$  
△ $\text{unknown}(m) − \text{unknown}(n)$
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: $m \leftarrow 1;\quad\triangleright\text{unknown}(m) - \text{unknown}(n)$
2: while $n > 1$ do
3: \hspace{0.5cm} $m \leftarrow m \times n;\quad\triangleright\text{unknown}(m) - \text{unknown}(n)$
4: \hspace{0.5cm} $n \leftarrow n - 1\quad\triangleright\text{unknown}(m) - \text{unknown}(n)$
5: end while
6: stop\quad\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.
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.
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,*
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.
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 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.
This allows us to determine the predecessors `pred` and successors `succ` of each statement, e.g. `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 (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:

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

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

\[
\text{RD}_{\text{exit}}(p) = (\text{RD}_{\text{entry}}(p) \setminus \text{kill}_{\text{RD}}(p)) \cup \text{gen}_{\text{RD}}(p)
\]
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: $RD_{init} = \{ (x, ?) | x \text{ is a variable in the program} \}$

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

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

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

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

\[ m \leftarrow 2 \]^1; \\
\textbf{while} \ [n > 1]^2 \ \textbf{do} \\
\quad [m \leftarrow m \times n]^3; \\
\quad [n \leftarrow n - 1]^4 \\
\textbf{end while} \\
[\text{stop}]^5
For our initial program fragment

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

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

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

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

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

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

- (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
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
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.
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

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

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

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

\[ \text{if } \left\{ \begin{array}{l}
y \in \text{FV}(a) \land (y, ?) \notin RD_{\text{entry}}(\ell) \land \\
\forall (y', \ell') \in RD_{\text{entry}}(\ell) : \\
y' = y \Rightarrow [\ldots]^\ell' = [ y := n ]^\ell'
\end{array} \right. \]

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

\[ \text{if } \left\{ \begin{array}{l}
\text{FV}(a) = \emptyset \land a \text{ is not constant} \land \\
a \text{ evaluates to } n
\end{array} \right. \]
Constant Folding II

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

\[
\begin{align*}
R D & \vdash S_1 \triangleright S_1' \\
R D & \vdash S_1; S_2 \triangleright S_1'; S_2
\end{align*}
\]

\[
\begin{align*}
R D & \vdash S_2 \triangleright S_2' \\
R D & \vdash S_1; S_2 \triangleright S_1'; S_2'
\end{align*}
\]
Constant Folding II

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

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

\[
\begin{align*}
    x &:= 10^1; \\
    y &:= x + 10^2; \\
    z &:= y + 10^3
\end{align*}
\]
An Example

To illustrate the use of the transformation consider:

\[
\begin{align*}
  & x := 10 \quad \text{\(1\)}; \\
  & y := x + 10 \quad \text{\(2\)}; \\
  & z := y + 10 \quad \text{\(3\)}
\end{align*}
\]

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

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

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

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

We can continue this kind of transformation and obtain:
Transformation

We can continue this kind of transformation and obtain:

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

We can continue this kind of transformation and obtain:

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

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

\[ RD \vdash \begin{array}{l}
[ x := 10 ]^1; [ y := x + 10 ]^2; [ z := y + 10 ]^3 \\
\triangleright [ x := 10 ]^1; [ y := 10 + 10 ]^2; [ z := y + 10 ]^3 \\
\triangleright [ x := 10 ]^1; [ y := 20 ]^2; [ z := y + 10 ]^3
\end{array} \]
Transformation

We can continue this kind of transformation and obtain:

\[ RD \vdash \begin{array}{l}
[ x := 10 ]^1; [ y := x + 10 ]^2; [ z := y + 10 ]^3 \\
\updownarrow [ x := 10 ]^1; [ y := 10 + 10 ]^2; [ z := y + 10 ]^3 \\
\updownarrow [ x := 10 ]^1; [ y := 20 ]^2; [ z := y + 10 ]^3 \\
\updownarrow [ x := 10 ]^1; [ y := 20 ]^2; [ z := 20 + 10 ]^3
\end{array} \]

after which no more steps are possible.
We can continue this kind of transformation and obtain:

\[ RD \vdash [ x := 10 ]^1; [ y := x + 10 ]^2; [ z := y + 10 ]^3 \]
\[ \uparrow [ x := 10 ]^1; [ y := 10 + 10 ]^2; [ z := y + 10 ]^3 \]
\[ \uparrow [ x := 10 ]^1; [ y := 20 ]^2; [ z := y + 10 ]^3 \]
\[ \uparrow [ x := 10 ]^1; [ y := 20 ]^2; [ z := 20 + 10 ]^3 \]
\[ \uparrow [ x := 10 ]^1; [ y := 20 ]^2; [ z := 30 ]^3 \]
We can continue this kind of transformation and obtain:

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

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

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

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

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

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.

\[ RD \vdash S_1 \triangleright S_2 \triangleright \ldots \triangleright S_n. \]
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 \textit{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 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.
Correctness

Any Program Analysis should be:

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

For example, why not consider in RD
entry (2) = RD exit (1) ∩ RD exit (4)
instead of RD
entry (2) = RD exit (1) ∪ 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.
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**.
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 $RD_{entry}(2) = RD_{exit}(1) \cup RD_{exit}(4)$. 
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 $0.\dot{9} = 0.99999\ldots = 1$?
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 $\mathbb{R}$. 

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 \(\mathbb{R}\). 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 $\lceil 0.\dot{9} \rceil = \lim(0.9, 0.99, 0.999, \ldots) = 1 = \lceil 1 \rceil$.

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