

Software Correctness: The Construction of Correct Software

Tracing Facts

Stefan Hallerstede (sha@ece.au.dk)

Carl Peter Leslie Schultz (cshultz@ece.au.dk)

John Hatcliff (Kansas State University)

Robby (Kansas State University)

Tracing Values
oooooooo

Tracing Facts
oooooooooooo

Tracing Facts with Logika
oooooooooooooo

Mutable Variables
ooooooooooooooo

Programs as Facts
oooo

Test Case Derivation
oooo

Symbolic Execution
oooo

Summary
oo

Tracing Values

Example

Discussion

Tracing Facts

Example

Discussion

Tracing Facts with Logika

Mutable Variables

Programs as Facts

Test Case Derivation

Symbolic Execution

Summary

Tracing Values
●○○○○○○

Tracing Facts
○○○○○○○○○○

Tracing Facts with Logika
○○○○○○○○○○○○

Mutable Variables
○○○○○○○○○○○○○○

Programs as Facts
○○○○

Test Case Derivation
○○○○

Symbolic Execution
○○○○

Summary
○○

Tracing Values

Example

Discussion

Tracing Facts

Example

Discussion

Tracing Facts with Logika

Mutable Variables

Programs as Facts

Test Case Derivation

Symbolic Execution

Summary



Execution as Calculation

- A common way to consider a program is to trace its execution following the values that variables take at different times

Execution as Calculation

- A common way to consider a program is to trace its execution following the values that variables take at different times
- We do this when debugging programs, where
 - we predict the values of variables
 - and observe deviations

Execution as Calculation

- A common way to consider a program is to trace its execution following the values that variables take at different times
- We do this when debugging programs, where
 - we predict the values of variables
 - and observe deviations
- This means,
 - we calculate expected values
 - and compare them to those produced by program execution

Execution as Calculation

- A common way to consider a program is to trace its execution following the values that variables take at different times
- We do this when debugging programs, where
 - we predict the values of variables
 - and observe deviations
- This means,
 - we calculate expected values
 - and compare them to those produced by program execution
- We can follow a similar approach directly without executing a program
 - stating expected values by asserting them
 - and compare them to those produced by calculation

Execution as Calculation

- A common way to consider a program is to trace its execution following the values that variables take at different times
- We do this when debugging programs, where
 - we predict the values of variables
 - and observe deviations
- This means,
 - we calculate expected values
 - and compare them to those produced by program execution
- We can follow a similar approach directly without executing a program
 - stating expected values by asserting them
 - and compare them to those produced by calculation
- Let's look at an example

Example: Execution as Calculation

```
val m: Z = 3
val n: Z = 5
val z: Z = m + n
val y: Z = z - n
val x: Z = z - y
assert(x == 5 & y == 3)
```

- The program above initialises variable `m` to 3 and variable `n` to 5

Example: Execution as Calculation

```
val m: Z = 3
val n: Z = 5
val z: Z = m + n
val y: Z = z - n
val x: Z = z - y
assert(x == 5 & y == 3)
```

- The program above initialises variable `m` to 3 and variable `n` to 5
- It asserts that variable `x` equals 5 and variable `y` to 3 at the end

Example: Execution as Calculation

```
val m: Z = 3
val n: Z = 5
val z: Z = m + n
val y: Z = z - n
val x: Z = z - y
assert(x == 5 & y == 3)
```

- The program above initialises variable `m` to 3 and variable `n` to 5
- It asserts that variable `x` equals 5 and variable `y` to 3 at the end
- Instead of using intermediate assertions (as done in the last lecture)
we state interspersed comments which variable values we can **deduce**

Example: Execution as Calculation

```
val m: Z = 3
val n: Z = 5
val z: Z = m + n
val y: Z = z - n
val x: Z = z - y
assert(x == 5 & y == 3)
```

- The program above initialises variable `m` to 3 and variable `n` to 5
- It asserts that variable `x` equals 5 and variable `y` to 3 at the end
- Instead of using intermediate assertions (as done in the last lecture)
we state interspersed comments which variable values we can **deduce**
- We insert lines

```
// deduce v == e & w == f & ...
```

specifying the deduced value `e` of variable `v`, `f` of variable `w` and so on

Example: Execution as Calculation

```
val m: Z = 3
// deduce m == 3
val n: Z = 5
// deduce m == 3 & n == 5
val z: Z = m + n
// deduce m == 3 & n == 5 & z == 8
val y: Z = z - n
// deduce m == 3 & n == 5 & z == 8 & y == 3
val x: Z = z - y
// deduce m == 3 & n == 5 & z == 8 & y == 3 & x == 5
assert(x == 5 & y == 3)
```

- All variables are assigned only once
- Hence, we only need to trace the value of each “new” variable

Example: Execution as Calculation

```
val m: Z = 3
// deduce m == 3
val n: Z = 5
// deduce m == 3 & n == 5
val z: Z = m + n
// deduce m == 3 & n == 5 & z == 8
val y: Z = z - n
// deduce m == 3 & n == 5 & z == 8 & y == 3
val x: Z = z - y
// deduce m == 3 & n == 5 & z == 8 & y == 3 & x == 5
assert(x == 5 & y == 3)
```

- All variables are assigned only once
- Hence, we only need to trace the value of each “new” variable

Example: Execution as Calculation

```
val m: Z = 3
// deduce m == 3
val n: Z = 5
// deduce m == 3 & n == 5
val z: Z = m + n
// deduce m == 3 & n == 5 & z == 8
val y: Z = z - n
// deduce m == 3 & n == 5 & z == 8 & y == 3
val x: Z = z - y
// deduce m == 3 & n == 5 & z == 8 & y == 3 & x == 5
assert(x == 5 & y == 3)
```

- All variables are assigned only once
- Hence, we only need to trace the value of each “new” variable

Example: Execution as Calculation

```
val m: Z = 3
// deduce m == 3
val n: Z = 5
// deduce m == 3 & n == 5
val z: Z = m + n
// deduce m == 3 & n == 5 & z == 8
val y: Z = z - n
// deduce m == 3 & n == 5 & z == 8 & y == 3
val x: Z = z - y
// deduce m == 3 & n == 5 & z == 8 & y == 3 & x == 5
assert(x == 5 & y == 3)
```

- All variables are assigned only once
- Hence, we only need to trace the value of each “new” variable

Example: Execution as Calculation

```
val m: Z = 3
// deduce m == 3
val n: Z = 5
// deduce m == 3 & n == 5
val z: Z = m + n
// deduce m == 3 & n == 5 & z == 8
val y: Z = z - n
// deduce m == 3 & n == 5 & z == 8 & y == 3
val x: Z = z - y
// deduce m == 3 & n == 5 & z == 8 & y == 3 & x == 5
assert(x == 5 & y == 3)
```

- All variables are assigned only once
- Hence, we only need to trace the value of each “new” variable

Example: Execution as Calculation

```
val m: Z = 3
// deduce m == 3
val n: Z = 5
// deduce m == 3 & n == 5
val z: Z = m + n
// deduce m == 3 & n == 5 & z == 8
val y: Z = z - n
// deduce m == 3 & n == 5 & z == 8 & y == 3
val x: Z = z - y
// deduce m == 3 & n == 5 & z == 8 & y == 3 & x == 5
assert(x == 5 & y == 3)
```

- All variables are assigned only once
- Hence, we only need to trace the value of each “new” variable

Example: Execution as Calculation

```
val m: Z = 3
// deduce m == 3
val n: Z = 5
// deduce m == 3 & n == 5
val z: Z = m + n
// deduce m == 3 & n == 5 & z == 8
val y: Z = z - n
// deduce m == 3 & n == 5 & z == 8 & y == 3
val x: Z = z - y
// deduce m == 3 & n == 5 & z == 8 & y == 3 & x == 5
assert(x == 5 & y == 3)
```

- The calculation yields `y == 3 & x == 5`

Example: Execution as Calculation

```
val m: Z = 3
// deduce m == 3
val n: Z = 5
// deduce m == 3 & n == 5
val z: Z = m + n
// deduce m == 3 & n == 5 & z == 8
val y: Z = z - n
// deduce m == 3 & n == 5 & z == 8 & y == 3
val x: Z = z - y
// deduce m == 3 & n == 5 & z == 8 & y == 3 & x == 5
assert(x == 5 & y == 3)
```

- The calculation yields `y == 3 & x == 5`
- It confirms the expected values stated in the final assertion

Discussion

- Starting from concrete values,
we have calculated the values of the variables during the execution of the program

Discussion

- Starting from concrete values,
we have calculated the values of the variables during the execution of the program
- This is easy!

Discussion

- Starting from concrete values,
we have calculated the values of the variables during the execution of the program
- This is easy!
- We have documented the values that we “know” by writing *// deduce ...*

Discussion

- Starting from concrete values,
we have calculated the values of the variables during the execution of the program
- This is easy!
- We have documented the values that we “know” by writing *// deduce ...*
- We have shown that the program is correct for the provided values

Discussion

- Starting from concrete values,
we have calculated the values of the variables during the execution of the program
- This is easy!
- We have documented the values that we “know” by writing *// deduce ...*
- We have shown that the program is correct for the provided values
- What if we wouldn’t know the initial values of variables *m* and *n*

Discussion

- Starting from concrete values,
we have calculated the values of the variables during the execution of the program
- This is easy!
- We have documented the values that we “know” by writing *// deduce ...*
- We have shown that the program is correct for the provided values
- What if we wouldn’t know the initial values of variables *m* and *n*
- In Slang we can express this by writing

```
val m: Z = randomInt()  
val n: Z = randomInt()
```

Discussion

- Starting from concrete values,
we have calculated the values of the variables during the execution of the program
- This is easy!
- We have documented the values that we “know” by writing *// deduce ...*
- We have shown that the program is correct for the provided values
- What if we wouldn’t know the initial values of variables `m` and `n`
- In Slang we can express this by writing

```
val m: Z = randomInt()  
val n: Z = randomInt()
```

- Function `randomInt()` specifies that an arbitrary integer value is chosen

Execution as Calculation Limitations

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- The method for tracing values by calculation does not work if the specific values of the variables are not known

Execution as Calculation Limitations

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- The method for tracing values by calculation does not work if the specific values of the variables are not known
- It is also not enough to limit deductions to specific variable values

Execution as Calculation Limitations

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- The method for tracing values by calculation does not work if the specific values of the variables are not known
- It is also not enough to limit deductions to specific variable values
- We need to trace more general kinds of **facts** that constrain possible values of variables

Execution as Calculation Limitations

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- The method for tracing values by calculation does not work if the specific values of the variables are not known
- It is also not enough to limit deductions to specific variable values
- We need to trace more general kinds of **facts** that constrain possible values of variables
- A specific variable value is just a special kind of fact that constrains a variable to one value

Execution as Calculation Limitations

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- The method for tracing values by calculation does not work if the specific values of the variables are not known
- It is also not enough to limit deductions to specific variable values
- We need to trace more general kinds of **facts** that constrain possible values of variables
- A specific variable value is just a special kind of fact that constrains a variable to one value
- Let's consider this example in more detail

Tracing Values
oooooooo

Tracing Facts
●oooooooooooo

Tracing Facts with Logika
oooooooooooo

Mutable Variables
ooooooooooooooo

Programs as Facts
oooo

Test Case Derivation
oooo

Symbolic Execution
oooo

Summary
oo

Tracing Values
Example
Discussion

Tracing Facts
Example
Discussion

Tracing Facts with Logika

Mutable Variables

Programs as Facts

Test Case Derivation

Symbolic Execution

Summary

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- There's nothing to deduce from the first two assignments except that `m` and `n` have arbitrary values

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- There's nothing to deduce from the first two assignments except that `m` and `n` have arbitrary values
- The first "interesting" fact that we can deduce follows the assignment to `z`

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- There's nothing to deduce from the first two assignments except that `m` and `n` have arbitrary values
- The first "interesting" fact that we can deduce follows the assignment to `z`
- After this assignment `z` must equal `m + n`

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- There's nothing to deduce from the first two assignments except that `m` and `n` have arbitrary values
- The first "interesting" fact that we can deduce follows the assignment to `z`
- After this assignment `z` must equal `m + n`
- We deduce `z == m + n` corresponding directly to the assignment `z = m + n`

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- There's nothing to deduce from the first two assignments except that `m` and `n` have arbitrary values
- The first "interesting" fact that we can deduce follows the assignment to `z`
- After this assignment `z` must equal `m + n`
- We deduce `z == m + n` corresponding directly to the assignment `z = m + n`
- Let's insert a comment introducing this fact

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- We can see immediately that there is such a fact directly deducible from each assignment

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- We can see immediately that there is such a fact directly deducible from each assignment
- Let's add those

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
// deduce y == z - n  
val x: Z = z - y  
// deduce x == z - y  
assert(x == n & y == m)
```

- However, this is not yet enough in order to deduce `x == n` and `y == m`

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
// deduce y == z - n  
val x: Z = z - y  
// deduce x == z - y  
assert(x == n & y == m)
```

- However, this is not yet enough in order to deduce `x == n` and `y == m`
- Because each variable is assigned only once, we can use “old facts”

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
// deduce y == z - n  
val x: Z = z - y  
// deduce x == z - y  
assert(x == n & y == m)
```

- However, this is not yet enough in order to deduce `x == n` and `y == m`
- Because each variable is assigned only once, we can use “old facts”
- Observe, that the variable `z` is referred to after the assignments to `y` and `x`

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
// deduce y == z - n  
val x: Z = z - y  
// deduce x == z - y  
assert(x == n & y == m)
```

- However, this is not yet enough in order to deduce `x == n` and `y == m`
- Because each variable is assigned only once, we can use “old facts”
- Observe, that the variable `z` is referred to after the assignments to `y` and `x`
- We can use the fact `z == m + n` there

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
// deduce z == m + n  
// deduce y == z - n  
val x: Z = z - y  
// deduce z == m + n  
// deduce x == z - y  
assert(x == n & y == m)
```

- Now, we join facts like `z == m + n` and `y == z - n`

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
// deduce z == m + n  
// deduce y == z - n  
val x: Z = z - y  
// deduce z == m + n  
// deduce x == z - y  
assert(x == n & y == m)
```

- Now, we join facts like $z == m + n$ and $y == z - n$
- From this we can deduce $y == (m + n) - n$

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
// deduce z == m + n  
// deduce y == z - n  
val x: Z = z - y  
// deduce z == m + n  
// deduce x == z - y  
assert(x == n & y == m)
```

- Now, we join facts like $z == m + n$ and $y == z - n$
- From this we can deduce $y == (m + n) - n$
- We deduce further $y == m + (n - n)$, and further $y == m + 0$

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
// deduce z == m + n  
// deduce y == z - n  
val x: Z = z - y  
// deduce z == m + n  
// deduce x == z - y  
assert(x == n & y == m)
```

- Now, we join facts like $z == m + n$ and $y == z - n$
- From this we can deduce $y == (m + n) - n$
- We deduce further $y == m + (n - n)$, and further $y == m + 0$
- Thus, $y == m$

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
// deduce z == m + n  
// deduce y == m  
val x: Z = z - y  
// deduce z == m + n  
// deduce x == z - y  
assert(x == n & y == m)
```

- The fact `y == m` is not affected by the assignment to `x`
- Hence, it remains true after that assignment

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
// deduce z == m + n  
// deduce y == m  
val x: Z = z - y  
// deduce z == m + n  
// deduce y == m  
// deduce x == z - y  
assert(x == n & y == m)
```

- From the joint facts $z == m + n$ and $y == m$ and $x == z - y$ we deduce $x == (m + n) - y$, further $x == (m + n) - m$, and further $x == (m - m) + n$, and $x == 0 + n$, thus $x == n$

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
// deduce z == m + n  
// deduce y == m  
val x: Z = z - y  
// deduce z == m + n  
// deduce y == m  
// deduce x == n  
assert(x == n & y == m)
```

- From the joint facts $z == m + n$ and $y == m$ and $x == z - y$ we deduce $x == (m + n) - y$, further $x == (m + n) - m$, and further $x == (m - m) + n$, and $x == 0 + n$, thus $x == n$

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n      (consequence of assignment)  
val y: Z = z - n  
// deduce z == m + n      (old fact)  
// deduce y == m          (proof by algebra)  
val x: Z = z - y  
// deduce z == m + n      (old fact)  
// deduce y == m          (old fact)  
// deduce x == n          (proof by algebra)  
assert(x == n & y == m)
```

- Facts that we use to demonstrate program correctness come from different sources as indicated above

Example: Deducing Facts for Immutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n      (consequence of assignment)  
val y: Z = z - n  
// deduce z == m + n      (old fact)  
// deduce y == m          (proof by algebra)  
val x: Z = z - y  
// deduce z == m + n      (old fact)  
// deduce y == m          (old fact)  
// deduce x == n          (proof by algebra)  
assert(x == n & y == m)
```

- Facts that we use to demonstrate program correctness come from different sources as indicated above
- Knowledge about the program is gathered and increased by inferring new facts

Tracing Values
○○○○○○○

Tracing Facts
○○○○○○○○●

Tracing Facts with Logika
○○○○○○○○○○○○

Mutable Variables
○○○○○○○○○○○○○○

Programs as Facts
○○○○

Test Case Derivation
○○○○

Symbolic Execution
○○○○

Summary
○○

Discussion

- We have generalised the approach of tracing values in programs to tracing facts

Discussion

- We have generalised the approach of tracing values in programs to tracing facts
- This has permitted us to demonstrate program correctness
independently of variables' initial values

Discussion

- We have generalised the approach of tracing values in programs to tracing facts
- This has permitted us to demonstrate program correctness **independently** of variables' initial values
- Without much difficulty we have attained a much more powerful method to verify programs

Discussion

- We have generalised the approach of tracing values in programs to tracing facts
- This has permitted us to demonstrate program correctness **independently** of variables' initial values
- Without much difficulty we have attained a much more powerful method to verify programs
- We would expect that the maths that we have applied could also be carried out **automatically**

Discussion

- We have generalised the approach of tracing values in programs to tracing facts
- This has permitted us to demonstrate program correctness **independently** of variables' initial values
- Without much difficulty we have attained a much more powerful method to verify programs
- We would expect that the maths that we have applied could also be carried out **automatically**
- Let's look at the program in Logika

Tracing Values
oooooooo

Tracing Facts
oooooooooooo

Tracing Facts with Logika
●oooooooooooo

Mutable Variables
oooooooooooooooo

Programs as Facts
oooo

Test Case Derivation
oooo

Symbolic Execution
oooo

Summary
oo

Tracing Values
Example
Discussion

Tracing Facts
Example
Discussion

Tracing Facts with Logika

Mutable Variables

Programs as Facts

Test Case Derivation

Symbolic Execution

Summary

Restating the Program with Deductions in Logika

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
// deduce z == m + n  
val y: Z = z - n  
// deduce z == m + n  
// deduce y == m  
val x: Z = z - y  
// deduce z == m + n  
// deduce y == m  
// deduce x == n  
assert (x == n & y == m)
```

- In order to have Logika check the deductions they have to be uncommented and stated in Logika syntax

Restating the Program with Deductions in Logika

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
Deduce( |- (z == m + n))  
val y: Z = z - n  
Deduce( |- (z == m + n))  
Deduce( |- (y == m))  
val x: Z = z - y  
Deduce( |- (z == m + n))  
Deduce( |- (y == m))  
Deduce( |- (x == n))  
assert(x == n & y == m)
```

- That's easy!

Restating the Program with Deductions in Logika

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
Deduce( |- (z == m + n) )  
val y: Z = z - n  
Deduce( |- (z == m + n) )  
Deduce( |- (y == m) )  
val x: Z = z - y  
Deduce( |- (z == m + n) )  
Deduce( |- (y == m) )  
Deduce( |- (x == n) )  
assert(x == n & y == m)
```

- That's easy!
- We've simply put the facts inside Logika's deduce commands

Restating the Program with Deductions in Logika

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
Deduce( |- (z == m + n) )  
val y: Z = z - n  
Deduce( |- (z == m + n) )  
Deduce( |- (y == m) )  
val x: Z = z - y  
Deduce( |- (z == m + n) )  
Deduce( |- (y == m) )  
Deduce( |- (x == n) )  
assert(x == n & y == m)
```

- That's easy!
- We've simply put the facts inside Logika's deduce commands
- In order to tell Logika that it's supposed to check this, a premise needs to be added

Restating the Program with Deductions in Logika

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt()
val n: Z = randomInt()
val z: Z = m + n
Deduce( |- (z == m + n))
val y: Z = z - n
Deduce( |- (z == m + n))
Deduce( |- (y == m))
val x: Z = z - y
Deduce( |- (z == m + n))
Deduce( |- (y == m))
Deduce( |- (x == n))
assert(x == n & y == m)
```

- We add the comment and `import` at the top

Restating the Program with Deductions in Logika

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt()
val n: Z = randomInt()
val z: Z = m + n
Deduce(|- (z == m + n))
val y: Z = z - n
Deduce(|- (z == m + n))
Deduce(|- (y == m))
val x: Z = z - y
Deduce(|- (z == m + n))
Deduce(|- (y == m))
Deduce(|- (x == n))
assert(x == n & y == m)
```

- We add the comment and `import` at the top

Logika

The screenshot shows the Sireum IDE interface with the following details:

- Title Bar:** Example – swap_immutable.sc [Example]
- Project View:** Shows the project structure under "Example". The "swap_immutable.sc" file is selected.
- Editor:** Displays the code for "swap_immutable.sc". The code uses Sireum's Logika extension to define a function that swaps two integers without using a temporary variable. The code is annotated with icons representing different analysis or transformation steps.
- Toolbars and Status Bar:** Standard Java-style toolbars. The status bar at the bottom shows "3:1 LF UTF-8 2 spaces".
- Right Panel:** Shows notifications and tool tabs for ASMPugin, JDT AST, and Sireum.

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt()
val n: Z = randomInt()
val z: Z = m + n
Deduce(|- (z == m + n))
val y: Z = z - n
Deduce(|- (z == m + n))
Deduce(|- (y == m))
val x: Z = z - y
Deduce(|- (z == m + n))
Deduce(|- (y == m))
Deduce(|- (x == n))
assert(x == n & y == m)
```

The Program in the Sireum/Logika IVE

Logika

```
// #Sireum #Logika
import org.sireum._

Click to show some hints nt()

val n: Z = randomInt()
val z: Z = m + n
Deduce(l - (z == m + n))
val y: Z = z - n
Deduce(l - (z == m + n))
Deduce(l - (y == m))
val x: Z = z - y
Deduce(l - (z == m + n))
Deduce(l - (y == m))
Deduce(l - (x == n))
assert(x == n & y == m)
```

Clicking on the light bulb shows facts known at that program location

Logika

The screenshot shows the Logika IDE interface. The title bar says "Example – swap_immutable.sc [Example]". The left sidebar has "Project" and "Structure" tabs, with "Project" selected. The project tree shows a "Week1" folder containing "logika" (with "swap_immutable.sc" selected), ".gitattributes", ".gitignore", "logika-example.sc", "readme.md", and "script-example.sc". Below "Week1" are "External Libraries" and "Scratches and Consoles". The main editor area displays the contents of "swap_immutable.sc":

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt()
val n: Z = randomInt()
val z: Z = m + n
Deduce(|- (z == m + n))
val y: Z = z - n
Deduce(|- (z == m + n))
Deduce(|- (y == m))
val x: Z = z - y
Deduce(|- (z == m + n))
Deduce(|- (y == m))
Deduce(|- (x == n))
assert(x == n & y == m)
```

The code editor has syntax highlighting for Logika and Sireum keywords. The right side of the interface includes toolbars for "Run swap_immutable.sc", "Sireum: Output", "Console", and "Notifications". A vertical toolbar on the right lists "ASMPugin", "JDT AST", and "Sireum". The bottom status bar shows "4:1 LF UTF-8 2 spaces" and "Projector".

Initially there aren't any known facts

Logika

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt()
val n: Z = randomInt()

Deduce(|- (z == m + n))
val y: Z = z - n
Deduce(|- (z == m + n))
Deduce(|- (y == m))
val x: Z = z - y
Deduce(|- (z == m + n))
Deduce(|- (y == m))
Deduce(|- (x == n))
assert(x == n & y == m)
```

After the two assignments to `m == randomInt()` and `n == randomInt()` ...

Logika

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt()
val n: Z = randomInt()
val z: Z = m + n
Deduce(|- (z == m + n))
val y: Z = z - n
Deduce(|- (z == m + n))
Deduce(|- (y == m))
val x: Z = z - y
Deduce(|- (z == m + n))
Deduce(|- (y == m))
Deduce(|- (x == n))
assert(x == n & y == m)
```

... it is only known that `m` and `n` have arbitrary values

Logika

The screenshot shows the Logika IDE interface. The main window displays a code editor with the file `swap_immutable.sc`. The code is as follows:

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt()
val n: Z = randomInt()
val z: Z = m + n
Deduce(|- (z == m + n))
Click to show some hints
Deduce(|- (z == m + n))
Deduce(|- (y == m))
val x: Z = z - y
Deduce(|- (z == m + n))
Deduce(|- (y == m))
Deduce(|- (x == n))
assert(x == n & y == m)
```

To the right of the code editor is a sidebar titled "Sireum: Output Console". It contains a section with a green checkmark and the following text:

```
{  
    m == At[Z](".random", 0);  
    n == At[Z](".random", 1)  
}
```

The sidebar also includes tabs for "Notifications", "ASMPugin", "JDT AST", and "Sireum". The bottom of the interface shows a toolbar with icons for Version Control, Run, TODO, Problems, Terminal, ANTLR Preview, Tool Output, and Services. The status bar at the bottom indicates "6:1 LF UTF-8 2 spaces" and "Projector".

Before the assignment to `y` ...

Logika

The screenshot shows the Logika IDE interface. On the left, the Project view displays a file tree for a project named 'Week1' containing files like '.idea', 'bin', 'out', 'logika', 'swap_immutable.sc', '.gitattributes', '.gitignore', 'logika-example.sc', 'readme.md', 'script-example.sc', and 'External Libraries'. The 'swap_immutable.sc' file is selected. The main editor window shows the following Sireum script:

```
val m: Z = randomInt()
val n: Z = randomInt()
val z: Z = m + n
Deduce(|- (z == m + n))
val y: Z = z - n
Deduce(|- (z == m + n))
Deduce(|- (y == m))
val x: Z = z - y
Deduce(|- (z == m + n))
Deduce(|- (y == m))
Deduce(|- (x == n))
assert(x == n & y == m)
```

A vertical bar highlights the line 'val y: Z = z - n'. To the right of the editor is a 'Sireum' panel with tabs for 'Output' and 'Console'. Below the editor, the status bar shows file statistics: 8:1 LF, UTF-8, 2 spaces, and a red dot icon for the 'Projector'.

...it is also known that $z == m + n$

Tracing Values
oooooooo

Tracing Facts
oooooooooooo

Tracing Facts with Logika
oooooooooooo

Mutable Variables
●oooooooooooo

Programs as Facts
oooo

Test Case Derivation
oooo

Symbolic Execution
ooo

Summary
oo

Tracing Values
Example
Discussion

Tracing Facts
Example
Discussion

Tracing Facts with Logika

Mutable Variables

Programs as Facts

Test Case Derivation

Symbolic Execution

Summary

Mutable Variables

- So far,
we have reasoned about programs with **immutable** variables
that are only assigned a value once

Mutable Variables

- So far,
we have reasoned about programs with **immutable** variables
that are only assigned a value once
- This was helpful
 - to learn about how facts propagate through programs
 - to get a first impression of Logika

Mutable Variables

- So far,
we have reasoned about programs with **immutable** variables
that are only assigned a value once
- This was helpful
 - to learn about how facts propagate through programs
 - to get a first impression of Logika
- Next,
we consider **mutable** variables
that can be assigned a new value repeatedly

Mutable Variables

- So far,
we have reasoned about programs with **immutable** variables
that are only assigned a value once
- This was helpful
 - to learn about how facts propagate through programs
 - to get a first impression of Logika
- Next,
we consider **mutable** variables
that can be assigned a new value repeatedly
- As a consequence,
we need to distinguish **old** values from **new** values for the **same** variable

Mutable Variables

- So far,
we have reasoned about programs with **immutable** variables
that are only assigned a value once
- This was helpful
 - to learn about how facts propagate through programs
 - to get a first impression of Logika
- Next,
we consider **mutable** variables
that can be assigned a new value repeatedly
- As a consequence,
we need to distinguish **old** values from **new** values for the **same** variable
- Let's have a closer look

From Immutable to Mutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- This is the same program we have seen before

From Immutable to Mutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- This is the same program we have seen before
- We would like to rewrite the program in such a way that it swaps the values of variables `x` and `y` in-place

From Immutable to Mutable Variables

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
val z: Z = m + n  
val y: Z = z - n  
val x: Z = z - y  
assert(x == n & y == m)
```

- This is the same program we have seen before
- We would like to rewrite the program in such a way that it swaps the values of variables `x` and `y` in-place
- Mutable variables are declared with the keyword `var` instead of `val`

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m  
var y: Z = n  
x = x + y  
y = x - y  
x = x - y  
assert(x == n & y == m)
```

- In this program `x` is assigned three times and `y` two times

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m  
var y: Z = n  
x = x + y  
y = x - y  
x = x - y  
assert(x == n & y == m)
```

- In this program `x` is assigned three times and `y` two times
- Let's try our method for immutable variables

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m  
var y: Z = n  
x = x + y  
y = x - y  
x = x - y  
assert(x == n & y == m)
```

- In this program `x` is assigned three times and `y` two times
- Let's try our method for immutable variables
 - After the first assignment to `x` we would obtain the fact `x == m`

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m  
var y: Z = n  
x = x + y  
y = x - y  
x = x - y  
assert(x == n & y == m)
```

- In this program `x` is assigned three times and `y` two times
- Let's try our method for immutable variables
 - After the first assignment to `x` we would obtain the fact `x == m`
 - After the second assignment to `x` we would obtain the fact `x == x + y`

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m  
var y: Z = n  
x = x + y  
y = x - y  
x = x - y  
assert(x == n & y == m)
```

- In this program `x` is assigned three times and `y` two times
- Let's try our method for immutable variables
 - After the first assignment to `x` we would obtain the fact `x == m`
 - After the second assignment to `x` we would obtain the fact `x == x + y`
 - This is not right!

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m  
var y: Z = n  
x = x + y  
y = x - y  
x = x - y  
assert(x == n & y == m)
```

- In this program `x` is assigned three times and `y` two times
- Let's try our method for immutable variables
 - After the first assignment to `x` we would obtain the fact `x == m`
 - After the second assignment to `x` we would obtain the fact `x == x + y`
 - This is not right!
 - The second assignment refers to the old value of `x` on the right-hand side, relating to the fact `x == m`
 - The left-hand side of that assignment refers to the new value

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m  
var y: Z = n  
x = x + y  
y = x - y  
x = x - y  
assert(x == n & y == m)
```

- Let's label the mutable variables according to the order in which they are assigned looking backwards from the final assertion `x == n & y == m`

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m    // 0  
var y: Z = n    // 0  
x = x + y      // 1  
y = x - y      // 1  
x = x - y      // 2  
assert(x == n & y == m)
```

- Let's label the mutable variables according to the order in which they are assigned

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m    // 0  
var y: Z = n    // 0  
x = x + y      // 1  
y = x - y      // 1  
x = x - y      // 2  
assert(x == n & y == m)
```

- Let's label the mutable variables according to the order in which they are assigned
- We can refer to variables v labelled by n by means of the expression $\text{At}(v, n)$

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m    // 0  
var y: Z = n    // 0  
x = x + y      // 1  
y = x - y      // 1  
x = x - y      // 2  
assert(x == n & y == m)
```

- Let's label the mutable variables according to the order in which they are assigned
- We can refer to variables v labelled by n by means of the expression $\text{At}(v, n)$
- For the last assignment (with the largest label) we let $\text{At}(v, n) == v$

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m    // 0  
var y: Z = n    // 0  
x = x + y      // 1  
y = x - y      // 1  
x = x - y      // 2  
assert(x == n & y == m)
```

- Let's label the mutable variables according to the order in which they are assigned
- We can refer to variables v labelled by n by means of the expression $\text{At}(v, n)$
- For the last assignment (with the largest label) we let $\text{At}(v, n) == v$
- Now, we can write in the comment behind each assignment the fact we deduce from it

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m    // deduce At(x, 0) == m  
var y: Z = n    // deduce At(y, 0) == n  
x = x + y      // deduce At(x, 1) == At(x, 0) + At(y, 0)  
y = x - y      // deduce y == At(x, 1) - At(y, 0)  
x = x - y      // deduce x == At(x, 1) - y  
assert(x == n & y == m)
```

- Let's label the mutable variables according to the order in which they are assigned
- We can refer to variables v labelled by n by means of the expression $\text{At}(v, n)$
- For the last assignment (with the largest label) we let $\text{At}(v, n) == v$
- Now, we can write in the comment behind each assignment the fact we deduce from it

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m    // deduce At(x, 0) == m  
var y: Z = n    // deduce At(y, 0) == n  
x = x + y      // deduce At(x, 1) == At(x, 0) + At(y, 0)  
y = x - y      // deduce y == At(x, 1) - At(y, 0)  
x = x - y      // deduce x == At(x, 1) - y  
assert(x == n & y == m)
```

- Let's label the mutable variables according to the order in which they are assigned
- We can refer to variables v labelled by n by means of the expression $\text{At}(v, n)$
- For the last assignment (with the largest label) we let $\text{At}(v, n) == v$
- Now, we can write in the comment behind each assignment the fact we deduce from it
- The problem has disappeared and we can reason about the program as before

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m    // deduce At(x, 0) == m  
var y: Z = n    // deduce At(y, 0) == n  
x = x + y      // deduce At(x, 1) == At(x, 0) + At(y, 0)  
y = x - y      // deduce y == At(x, 1) - At(y, 0)  
x = x - y      // deduce x == At(x, 1) - y  
assert(x == n & y == m)
```

- Let's label the mutable variables according to the order in which they are assigned
- We can refer to variables v labelled by n by means of the expression $\text{At}(v, n)$
- For the last assignment (with the largest label) we let $\text{At}(v, n) == v$
- Now, we can write in the comment behind each assignment the fact we deduce from it
- The problem has disappeared and we can reason about the program as before
- We can apply this method to any location in a program, replacing variable up to that point

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m    // deduce At(x, 0) == m  
var y: Z = n    // deduce At(y, 0) == n  
x = x + y      // deduce At(x, 1) == At(x, 0) + At(y, 0)  
y = x - y      // deduce y == At(x, 1) - At(y, 0)  
x = x - y      // deduce x == At(x, 1) - y  
assert(x == n & y == m)
```

- Let's label the mutable variables according to the order in which they are assigned
- We can refer to variables v labelled by n by means of the expression $\text{At}(v, n)$
- For the last assignment (with the largest label) we let $\text{At}(v, n) == v$
- Now, we can write in the comment behind each assignment the fact we deduce from it
- The problem has disappeared and we can reason about the program as before
- We can apply this method to any location in a program, replacing variable up to that point
- Let's see how this looks in Logika

Logika

The screenshot shows the Logika IDE interface. On the left, the Project view displays a file structure for 'Week1' containing 'logika' and 'swap_immutable.sc'. The 'swap_immutable.sc' file is selected. The main code editor window shows the following pseudocode:

```
val m: Z = randomInt()
val n: Z = randomInt()
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

A yellow highlight covers the assignment line 'x = x + y'. To the right of the code editor is a 'Sireum' panel showing a checkmark and a block of Sireum code:

```
{  
    m == At[Z](".random", 0);  
    n == At[Z](".random", 1);  
    x == m;  
    y == n  
}
```

The bottom of the interface includes standard IDE navigation bars like Version Control, Run, TODO, Problems, Terminal, ANTLR Preview, Tool Output, and Services.

Before the second assignment to `x` all variables are referred to by their original name

Logika

The screenshot shows the Logika IDE interface. The title bar says "Example - swap Mutable.sc [Example]". The left sidebar has a "Project" view showing a file tree with ".idea", "bin", "out", and a "Week1" folder containing "logika", "smt2", "example.json", "example2.json", "swap_imutable.sc", and "swap Mutable.sc". The "swap Mutable.sc" file is selected. The main editor area contains the following Sireum code:

```
val m: Z = randomInt()
val n: Z = randomInt()
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

To the right of the editor is a "Sireum" panel with tabs for "Output" and "Console". Below the editor is a status bar showing "9:2 LF UTF-8 2 spaces".

After the second assignment to `x` the “old” `x` is referred to by `At(x, 0)`

Logika

The screenshot shows the Logika IDE interface. The title bar reads "Example - swap Mutable.sc [Example]". The code editor displays the following Sireum script:

```
val m: Z = randomInt()
val n: Z = randomInt()
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

The code editor has syntax highlighting for Sireum keywords (e.g., val, var, assert) and types (e.g., Z). A yellow selection bar highlights the entire code block. To the right of the code editor is a vertical toolbar with several tabs: Sireum, Output, Console, Notifications, ASMPugin, JDT AST, and Sireum. The "Sireum" tab is currently selected. Below the code editor is a navigation bar with links to Version Control, Run, TODO, Problems, Terminal, ANTLR Preview, Tool Output, and Services. At the bottom of the screen, there is a status bar showing the time (10:2), file encoding (LF, UTF-8), and spaces (2 spaces).

Continuing in this way all “old” variables are replaced ...

Logika

The screenshot shows the Logika IDE interface. On the left, the Project view displays a file tree for a 'Week1' project containing files like .idea, bin, out, Week1, logika, smt2, example.json, example2.json, swap_immutable.sc, and swap Mutable.sc. The swap Mutable.sc file is selected. The main area is a code editor with the following content:

```
Example > Week1 > swap Mutable.sc [Example]
swap Mutable.sc
import org.logika.com._

3
4
5
6
7
8
9
10
11
12

val m: Z = randomInt()
val n: Z = randomInt()
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

To the right of the code editor, a Sireum analysis window is open, showing the Pre-state at line 11 and Post-state at line 11. The Pre-state is empty. The Post-state contains the following assertions:

```
{
  m == At[Z](".random", 0);
  n == At[Z](".random", 1);
  At(x, 0) == m;
  At(y, 0) == n;
  At(x, 1) == At(x, 0) + At(y, 0);
  y == At(x, 1) - At(y, 0);
  x == At(x, 1) - y
}
```

The bottom of the interface shows standard IDE navigation bars: Version Control, Run, TODO, Problems, Terminal, ANTLR Preview, Tool Output, Services, and a status bar indicating 12:1 LF UTF-8 2 spaces.

... until we reach the final assertion

Example: Mutable Swapping

```
val m: Z = randomInt ()  
val n: Z = randomInt ()  
var x: Z = m  
var y: Z = n  
x = x + y  
y = x - y  
x = x - y  
assert(x == n & y == m)
```

- With the replacement method used in Logika we can add deductions to prove the assertion

Example: Mutable Swapping

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt()
val n: Z = randomInt()
var x: Z = m
var y: Z = n
x = x + y
Deduce(|- (At(x, 0) == m))
Deduce(|- (y == n))
Deduce(|- (x == At(x, 0) + y))
Deduce(|- (x == m + n))
```

```
y = x - y
Deduce(|- (x == m + n))
Deduce(|- (At(y, 0) == n))
Deduce(|- (y == x - At(y, 0)))
Deduce(|- (y == m))
x = x - y
Deduce(|- (At(x, 1) == m + n))
Deduce(|- (y == m))
Deduce(|- (x == At(x, 1) - y))
assert(x == n & y == m)
```

Tracing Values
oooooooo

Tracing Facts
oooooooooooo

Tracing Facts with Logika
oooooooooooooo

Mutable Variables
ooooooooooooooo

Programs as Facts
●oooo

Test Case Derivation
oooo

Symbolic Execution
oooo

Summary
oo

Tracing Values
Example
Discussion

Tracing Facts
Example
Discussion

Tracing Facts with Logika

Mutable Variables

Programs as Facts

Test Case Derivation

Symbolic Execution

Summary

Facts from Assignments

- We have observed that each assignment of the shape
 $v = e$ for a variable v and expression e
gives rise to a fact $\text{At}(v, n) == e_{old}$,
for some n ,
where e_{old} is e with all variables w replaced by their “old” values $\text{At}(w, m)$

Facts from Assignments

- We have observed that each assignment of the shape
 $v = e$ for a variable v and expression e
gives rise to a fact $\text{At}(v, n) == e_{old}$,
for some n ,
where e_{old} is e with all variables w replaced by their “old” values $\text{At}(w, m)$
- Thus, a program P
corresponds to the conjunction of the facts
originating from the sequence of assignments

Facts from Assignments

- We have observed that each assignment of the shape $v = e$ for a variable v and expression e gives rise to a fact $\text{At}(v, n) == e_{old}$, for some n , where e_{old} is e with all variables w replaced by their “old” values $\text{At}(w, m)$
- Thus, a program P corresponds to the conjunction of the facts originating from the sequence of assignments
- This means the program P itself is also just fact P_{fact} about which we can reason

Facts from Assignments

- We have observed that each assignment of the shape $v = e$ for a variable v and expression e gives rise to a fact $\text{At}(v, n) == e_{old}$, for some n , where e_{old} is e with all variables w replaced by their “old” values $\text{At}(w, m)$
- Thus, a program P corresponds to the conjunction of the facts originating from the sequence of assignments
- This means the program P itself is also just fact P_{fact} about which we can reason
- The components of P_{fact} have a one-to-one correspondence with the assignments of P

Example: Mutable Swapping

<code>val m: Z = randomInt()</code>	<code>m == At[Z](".random", 0)</code>	&
<code>val n: Z = randomInt()</code>	<code>n == At[Z](".random", 1)</code>	&
<code>var x: Z = m</code>	<code>At(x, 0) == m</code>	&
<code>var y: Z = n</code>	<code>At(y, 0) == n</code>	&
<code>x = x + y</code>	<code>At(x, 1) == At(x, 0) + At(y, 0)</code>	&
<code>y = x - y</code>	<code>y == At(x, 1) - At(y, 0)</code>	&
<code>x = x - y</code>	<code>x == At(x, 1) - y</code>	

- Each line of the program P corresponds to a fact and the program itself is the conjunction P_{fact} of those facts

Example: Mutable Swapping

<code>val m: Z = randomInt()</code>	<code>m == At[Z](".random", 0)</code>	&
<code>val n: Z = randomInt()</code>	<code>n == At[Z](".random", 1)</code>	&
<code>var x: Z = m</code>	<code>At(x, 0) == m</code>	&
<code>var y: Z = n</code>	<code>At(y, 0) == n</code>	&
<code>x = x + y</code>	<code>At(x, 1) == At(x, 0) + At(y, 0)</code>	&
<code>y = x - y</code>	<code>y == At(x, 1) - At(y, 0)</code>	&
<code>x = x - y</code>	<code>x == At(x, 1) - y</code>	

- Each line of the program P corresponds to a fact and the program itself is the conjunction P_{fact} of those facts
- The assertion `assert (x == n & y == m)` corresponds to a fact a , namely, `x == n & y == m`

Example: Mutable Swapping

<code>val m: Z = randomInt()</code>	<code>m == At[Z](".random", 0)</code>	&
<code>val n: Z = randomInt()</code>	<code>n == At[Z](".random", 1)</code>	&
<code>var x: Z = m</code>	<code>At(x, 0) == m</code>	&
<code>var y: Z = n</code>	<code>At(y, 0) == n</code>	&
<code>x = x + y</code>	<code>At(x, 1) == At(x, 0) + At(y, 0)</code>	&
<code>y = x - y</code>	<code>y == At(x, 1) - At(y, 0)</code>	&
<code>x = x - y</code>	<code>x == At(x, 1) - y</code>	

- Each line of the program P corresponds to a fact and the program itself is the conjunction P_{fact} of those facts
- The assertion `assert (x == n & y == m)` corresponds to a fact a , namely, `x == n & y == m`
- We can use it to ask different questions about the program

Tracing Values
oooooooo

Tracing Facts
oooooooooooo

Tracing Facts with Logika
oooooooooooooo

Mutable Variables
ooooooooooooooo

Programs as Facts
oooo●

Test Case Derivation
oooo

Symbolic Execution
oooo

Summary
oo

Analysing Programs

- We have three ways in which we can use P_{fact} to analyse program P

Analysing Programs

- We have three ways in which we can use P_{fact} to analyse program P
- Using $P_{fact} \vdash a$, we can **prove** that assertion a is true for all executions of P
- This is what we have done in Logika

Analysing Programs

- We have three ways in which we can use P_{fact} to analyse program P
- Using $P_{fact} \mid - a$, we can **prove** that assertion a is true for all executions of P
- This is what we have done in Logika
- Using $P_{fact} \& a$, we can search for values for which a is true
- This is the basis for generating **tests** (We have to remove assignments from `randomInt()` first)

Analysing Programs

- We have three ways in which we can use P_{fact} to analyse program P
- Using $P_{fact} \mid - a$, we can **prove** that assertion a is true for all executions of P
- This is what we have done in Logika
- Using $P_{fact} \& a$, we can search for values for which a is true
- This is the basis for generating **tests** (We have to remove assignments from `randomInt()` first)
- Using $P_{fact} \& !a$, we can search for values for which a is false
- This yields **counterexamples**, i.e., specific values for the variables of P that violate a
- If no such value can be found among all possible values of all variables, then $P_{fact} \mid - a$ must be true
- This technique is referred to as (bounded) **model checking**

Tracing Values
oooooooo

Tracing Facts
oooooooooooo

Tracing Facts with Logika
oooooooooooooo

Mutable Variables
ooooooooooooooo

Programs as Facts
oooo

Test Case Derivation
●oooo

Symbolic Execution
oooo

Summary
oo

Tracing Values
Example
Discussion

Tracing Facts
Example
Discussion

Tracing Facts with Logika

Mutable Variables

Programs as Facts

Test Case Derivation

Symbolic Execution

Summary

Example: Mutable Swapping

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- We bracket the program in an `assume-assert` contract

Example: Mutable Swapping

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- We bracket the program in an `assume-assert` contract
- Assuming the condition `m > 0 & y > 0` is true initially, the condition `x == n & y == m` must be true finally

Example: Mutable Swapping

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- We bracket the program in an `assume-assert` contract
- Assuming the condition `m > 0 & y > 0` is true initially, the condition `x == n & y == m` must be true finally
- We search for values of `m`, `n`, `x` and `y` that achieve this

Example: Mutable Swapping

assume($m > 0 \ \& \ n > 0$)	$m > 0 \ \& \ n > 0$	&
var $x : Z = m$	At($x, 0$) == m	&
var $y : Z = n$	At($y, 0$) == n	&
$x = x + y$	At($x, 1$) == At($x, 0$) + At($y, 0$)	&
$y = x - y$	$y == At(x, 1) - At(y, 0)$	&
$x = x - y$	$x == At(x, 1) - y$	&
assert($x == n \ \& \ y == m$)	$x == n \ \& \ y == m$	

- We bracket the program in an `assume-assert` contract
- Assuming the condition $m > 0 \ \& \ n > 0$ is true initially, the condition $x == n \ \& \ y == m$ must be true finally
- We search for values of m, n, x and y that achieve this

Example: Mutable Swapping

```
assume(m > 0 & n > 0)           m > 0 & n > 0 &
var x: Z = m                      At(x, 0) == m &
var y: Z = n                      At(y, 0) == n &
x = x + y                        At(x, 1) == At(x, 0) + At(y, 0) &
y = x - y                        y == At(x, 1) - At(y, 0) &
x = x - y                        x == At(x, 1) - y &
assert(x == n & y == m)          x == n & y == m
```

- We search for values of `m`, `n`, `x` and `y` that can be used for testing

Example: Mutable Swapping

```
assume(m > 0 & n > 0)           m > 0 & n > 0 &
var x: Z = m                      At(x, 0) == m &
var y: Z = n                      At(y, 0) == n &
x = x + y                         At(x, 1) == At(x, 0) + At(y, 0) &
y = x - y                         y == At(x, 1) - At(y, 0) &
x = x - y                         x == At(x, 1) - y &
assert(x == n & y == m)           x == n & y == m
```

- We search for values of `m`, `n`, `x` and `y` that can be used for testing
- E.g.,
input: `m == 1, n == 2`
output: `m == 1, n == 2, x == 2` and `y == 1`

Example: Mutable Swapping

```
assume(m > 0 & n > 0)           m > 0 & n > 0 &
var x: Z = m                      At(x, 0) == m &
var y: Z = n                      At(y, 0) == n &
x = x + y                         At(x, 1) == At(x, 0) + At(y, 0) &
y = x - y                         y == At(x, 1) - At(y, 0) &
x = x - y                         x == At(x, 1) - y &
assert(x == n & y == m)           x == n & y == m
```

- We search for values of `m`, `n`, `x` and `y` that can be used for testing
- E.g.,
 - input: `m == 1, n == 2`
 - output: `m == 1, n == 2, x == 2` and `y == 1`
- We will see more interesting uses of this during the course

Tracing Values
oooooooo

Tracing Facts
oooooooooooo

Tracing Facts with Logika
oooooooooooooo

Mutable Variables
ooooooooooooooo

Programs as Facts
oooo

Test Case Derivation
oooo

Symbolic Execution
●○○○

Summary
oo

Tracing Values

Example
Discussion

Tracing Facts

Example
Discussion

Tracing Facts with Logika

Mutable Variables

Programs as Facts

Test Case Derivation

Symbolic Execution

Summary

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- There's yet another way we can reason about this program

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- There's yet another way we can reason about this program
- We execute the program abstractly with **symbolic values** and **path conditions**

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- There's yet another way we can reason about this program
- We execute the program abstractly with **symbolic values** and **path conditions**
- Symbolic values record the modifications of the variables

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- There's yet another way we can reason about this program
- We execute the program abstractly with **symbolic values** and **path conditions**
- Symbolic values record the modifications of the variables
- Path conditions record the conditions that must be true to reach locations in the program

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- There's yet another way we can reason about this program
- We execute the program abstractly with **symbolic values** and **path conditions**
- Symbolic values record the modifications of the variables
- Path conditions record the conditions that must be true to reach locations in the program
- We record symbolic values in the tuple
 (m, n, x, y)
and the path condition as
 $(PC: \dots)$

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- There's yet another way we can reason about this program
- We execute the program abstractly with **symbolic values** and **path conditions**
- Symbolic values record the modifications of the variables
- Path conditions record the conditions that must be true to reach locations in the program
- We record symbolic values in the tuple
 (m, n, x, y)
and the path condition as
 $(PC: \dots)$
- Let's do this with the value swapping example

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

Using Facts as Values

```
assume(m > 0 & n > 0)

var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- Executing `assume(m > 0 & n > 0)` yields $(m: M, n: N), (PC: M > 0 \& N > 0)$

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- Executing `assume(m > 0 & n > 0)` yields $(m: M, n: N), (PC: M > 0 \& N > 0)$
- Executing `var x: Z = m` yields $(m: M, n: N, x: M), (PC: M > 0 \& N > 0)$

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- Executing `assume(m > 0 & n > 0)` yields $(m: M, n: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `var x: Z = m` yields $(m: M, n: N, x: M), (\text{PC}: M > 0 \& N > 0)$
- Executing `var y: Z = n` yields $(m: M, n: N, x: M, y: N), (\text{PC}: M > 0 \& N > 0)$

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- Executing `assume(m > 0 & n > 0)` yields $(m: M, n: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `var x: Z = m` yields $(m: M, n: N, x: M), (\text{PC}: M > 0 \& N > 0)$
- Executing `var y: Z = n` yields $(m: M, n: N, x: M, y: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `x = x + y` yields $(m: M, n: N, x: M + N, y: N), (\text{PC}: M > 0 \& N > 0)$

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- Executing `assume(m > 0 & n > 0)` yields $(m: M, n: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `var x: Z = m` yields $(m: M, n: N, x: M), (\text{PC}: M > 0 \& N > 0)$
- Executing `var y: Z = n` yields $(m: M, n: N, x: M, y: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `x = x + y` yields $(m: M, n: N, x: M + N, y: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `y = x - y` yields $(m: M, n: N, x: M + N, y: M), (\text{PC}: M > 0 \& N > 0)$

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- Executing `assume(m > 0 & n > 0)` yields $(m: M, n: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `var x: Z = m` yields $(m: M, n: N, x: M), (\text{PC}: M > 0 \& N > 0)$
- Executing `var y: Z = n` yields $(m: M, n: N, x: M, y: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `x = x + y` yields $(m: M, n: N, x: M + N, y: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `y = x - y` yields $(m: M, n: N, x: M + N, y: M), (\text{PC}: M > 0 \& N > 0)$
- Executing `x = x - y` yields $(m: M, n: N, x: N, y: M), (\text{PC}: M > 0 \& N > 0)$

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- Executing `assume(m > 0 & n > 0)` yields $(m: M, n: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `var x: Z = m` yields $(m: M, n: N, x: M), (\text{PC}: M > 0 \& N > 0)$
- Executing `var y: Z = n` yields $(m: M, n: N, x: M, y: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `x = x + y` yields $(m: M, n: N, x: M + N, y: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `y = x - y` yields $(m: M, n: N, x: M + N, y: M), (\text{PC}: M > 0 \& N > 0)$
- Executing `x = x - y` yields $(m: M, n: N, x: N, y: M), (\text{PC}: M > 0 \& N > 0)$
- Executing `assert(x == n & y == m)` yields
 $(m: M, n: N, x: N, y: M), (\text{PC}: M > 0 \& N > 0, N == N \& M == M)$

Using Facts as Values

```
assume(m > 0 & n > 0)
var x: Z = m
var y: Z = n
x = x + y
y = x - y
x = x - y
assert(x == n & y == m)
```

- Executing `assume(m > 0 & n > 0)` yields $(m: M, n: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `var x: Z = m` yields $(m: M, n: N, x: M), (\text{PC}: M > 0 \& N > 0)$
- Executing `var y: Z = n` yields $(m: M, n: N, x: M, y: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `x = x + y` yields $(m: M, n: N, x: M + N, y: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `y = x - y` yields The expression $M + N$ cannot be further simplified at this stage because M and N are uninterpreted symbolic constants.
- Executing `x = x - y` yields $(m: M, n: N, x: M, y: N), (\text{PC}: M > 0 \& N > 0)$
- Executing `assert(x == n & y == m)` yields $(m: M, n: N, x: N, y: M), (\text{PC}: M > 0 \& N > 0, N == N \& M == M)$

Tracing Values
oooooooo

Tracing Facts
oooooooooooo

Tracing Facts with Logika
oooooooooooooo

Mutable Variables
ooooooooooooooo

Programs as Facts
oooo

Test Case Derivation
oooo

Symbolic Execution
oooo

Summary
●○

Tracing Values

Example
Discussion

Tracing Facts

Example
Discussion

Tracing Facts with Logika

Mutable Variables

Programs as Facts

Test Case Derivation

Symbolic Execution

Summary



Summary

- We have looked at various ways of reasoning about programs
 - by tracing facts through programs
 - by considering programs as facts
 - by symbolic execution
- We have seen how this can be used to reason in different ways about programs
 - to prove assertions
 - to find counterexamples
 - to generate tests
- We will discuss this continually during the course as the programs become more and more challenging