Software Correctness:
The Construction of Correct Software

Tracing Facts

Stefan Hallerstede (sha@ece.au.dk)
Carl Peter Leslie Schultz (cschultz@ece.au.dk)

John Hatcliff (Kansas State University)
Robby (Kansas State University)

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 in 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 in 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

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

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

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

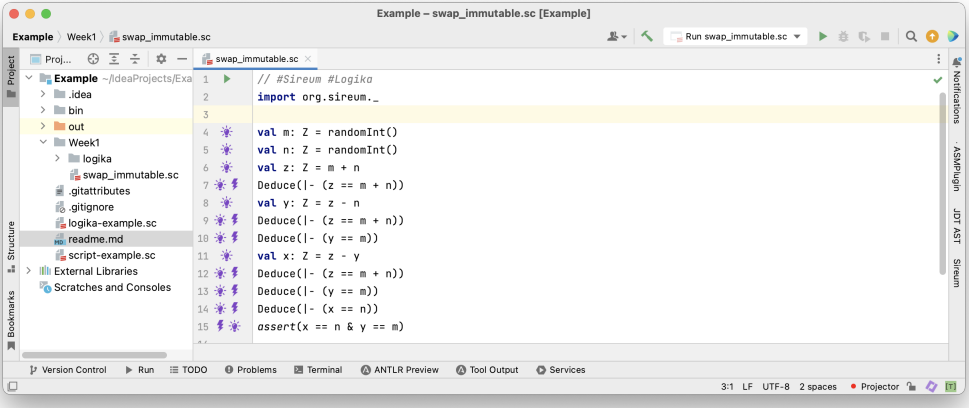
Restating the Program with Deductions in Logika

```
// #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)
```

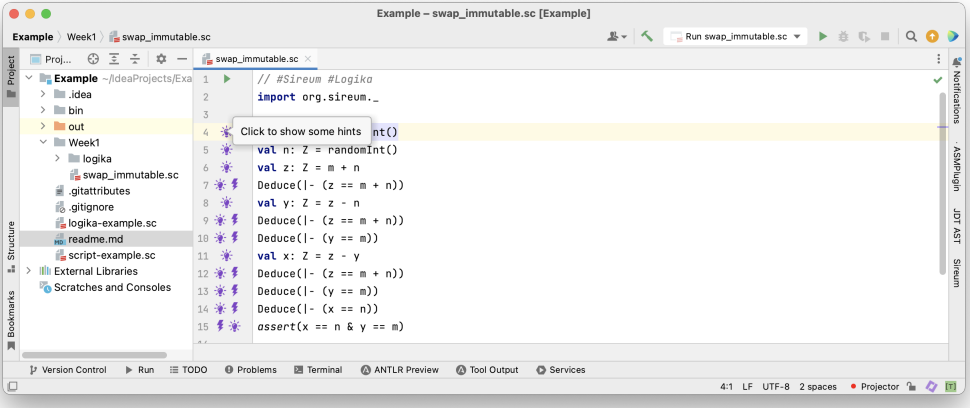
- We add the comment and **import** at the top

Logika



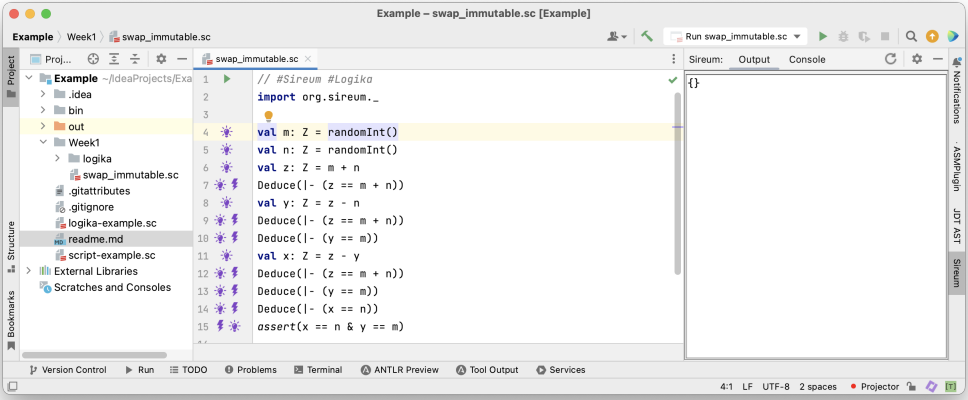
The Program in the Sireum/Logika IVE

Logika



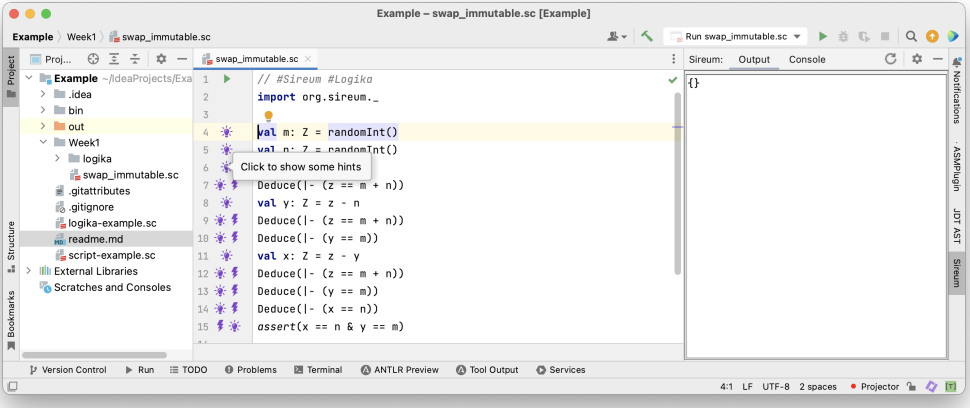
Clicking on the light bulb shows facts known at that program location

Logika



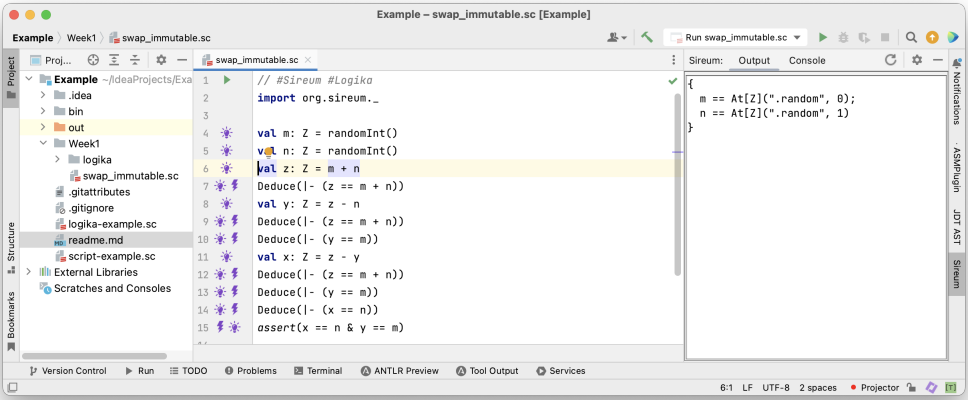
Initially there aren't any known facts

Logika



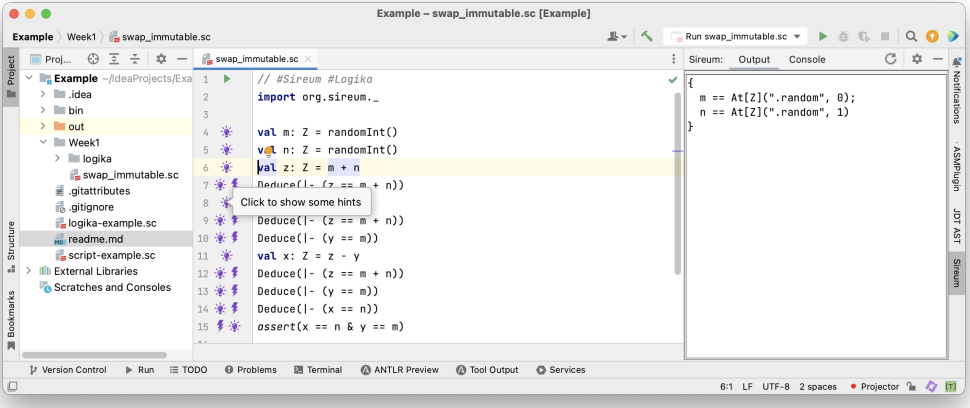
After the two assignments to `m == randomInt()` and `n == randomInt()` ...

Logika



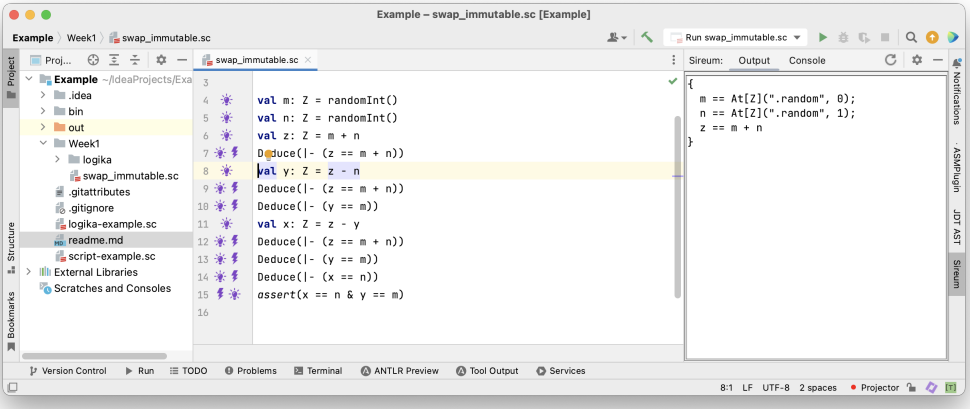
. . . it is only known that `m` and `n` have arbitrary values

Logika



Before the assignment to y . . .

Logika



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

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 though 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 though 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 though 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 though 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 `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 `At(`v`, `n`)`
- For the last assignment (with the largest label) we let `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 `At(v, n)`
- For the last assignment (with the largest label) we let `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 `At(v, n)`
- For the last assignment (with the largest label) we let `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 `At(v, n)`
- For the last assignment (with the largest label) we let `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)
```

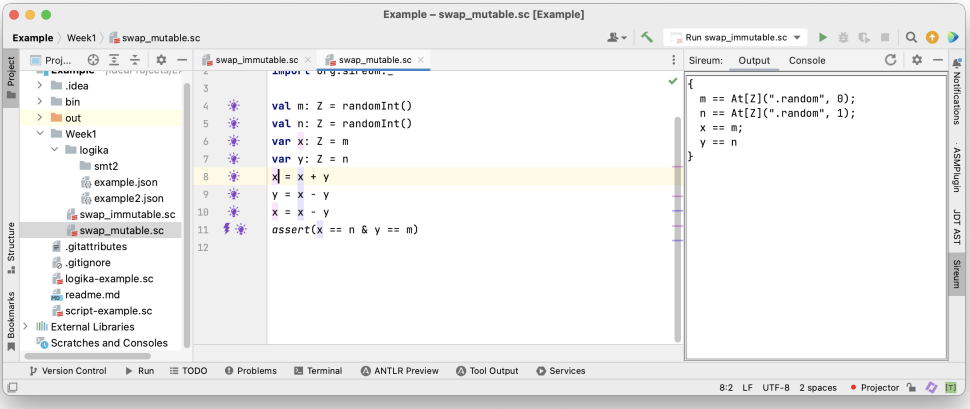
- Let's label the mutable variables according to the order in which they are assigned
- We can refer to variables v labelled by n by means of the expression `At(v, n)`
- For the last assignment (with the largest label) we let `At(v, n) == v`
- Now, we can write in the comment behind each assignment the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

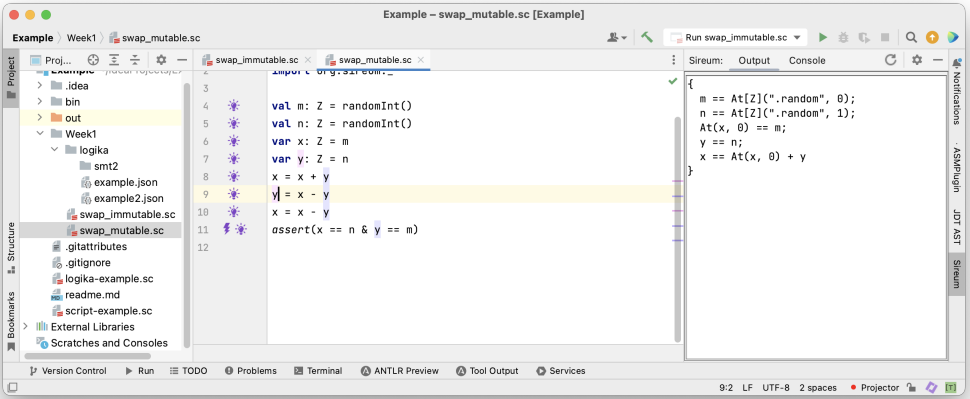
- Let's label the mutable variables according to the order in which they are assigned
- We can refer to variables v labelled by n by means of the expression `At(v, n)`
- For the last assignment (with the largest label) we let `At(v, n) == v`
- Now, we can write in the comment behind each assignment the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



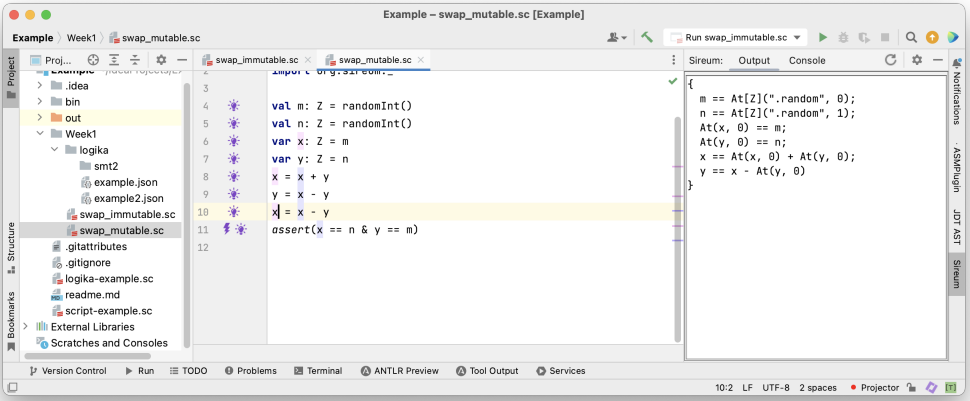
Before the second assignment to `x` all variables are referred to by their original name

Logika



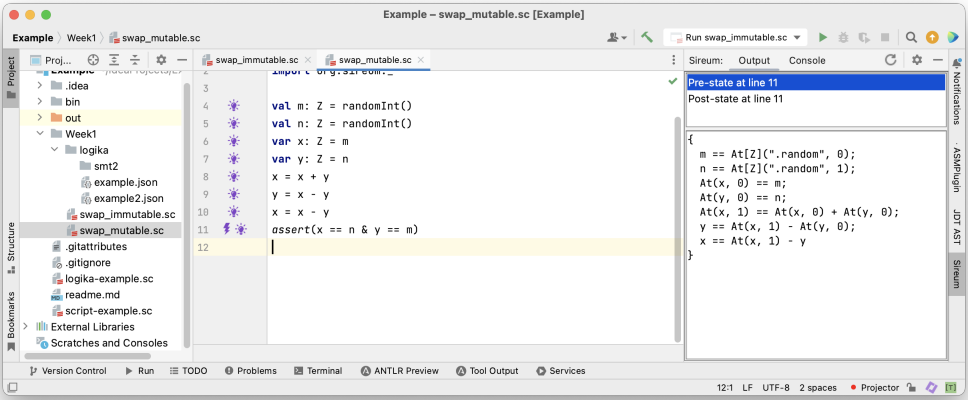
After the second assignment to `x` the "old" `x` is referred to by `At(x, 0)`

Logika



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

Logika



. . . 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)
```

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 `At(`v`,`n`)` `==` e_{old},
  for some n,
  where e_{old} is e with all variables w replaced by their "old" values `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 `At(`v`,`n`)` `==` e_{old},
  for some n,
  where e_{old} is e with all variables w replaced by their "old" values `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 `At(`v`,`n`)` `==` e_{old},
  for some n,
  where e_{old} is e with all variables w replaced by their "old" values `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 `At(`v`,`n`)` `==` e_{old},
  for some n,
  where e_{old} is e with all variables w replaced by their "old" values `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

```
val m: Z = randomInt()          m == At[Z](".random", 0)              &
val n: Z = randomInt()          n == At[Z](".random", 1)              &
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
```

- 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

```
val m: Z = randomInt()          m == At[Z](".random", 0)          &
val n: Z = randomInt()          n == At[Z](".random", 1)          &
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
```

- 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

```
val m: Z = randomInt()          m == At[Z](".random", 0)              &
val n: Z = randomInt()          n == At[Z](".random", 1)              &
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
```

- 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

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 $P_{fact} \mid\!- \; a$, 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} `|-` 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} `|-` 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} `|-` a must be true
- This technique is referred to as (bounded) **model checking**

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

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)
```

```
m > 0 & n > 0                                  &
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                              &
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

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: . . .)

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: . . .)
- 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`), (PC: `M > 0 & N > 0`)
- Executing `var x: Z = m` yields (`m: M`, `n: N`, `x: M`), (PC: `M > 0 & N > 0`)
- Executing `var y: Z = n` yields (`m: M`, `n: N`, `x: M`, `y: 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`)
- Executing `var y: Z = n` yields (`m: M`, `n: N`, `x: M`, `y: N`), (PC: `M > 0 & N > 0`)
- Executing `x = x + y` yields (`m: M`, `n: N`, `x: M + N`, `y: 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`)
- Executing `var y: Z = n` yields (`m: M, n: N, x: M, y: N`), (PC: `M > 0 & N > 0`)
- Executing `x = x + y` yields (`m: M, n: N, x: M + N, y: N`), (PC: `M > 0 & N > 0`)
- Executing `y = x - y` yields (`m: M, n: N, x: M + N, y: 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`), (PC: `M > 0 & N > 0`)
- Executing `var x: Z = m` yields (`m: M, n: N, x: M`), (PC: `M > 0 & N > 0`)
- Executing `var y: Z = n` yields (`m: M, n: N, x: M, y: N`), (PC: `M > 0 & N > 0`)
- Executing `x = x + y` yields (`m: M, n: N, x: M + N, y: N`), (PC: `M > 0 & N > 0`)
- Executing `y = x - y` yields (`m: M, n: N, x: M + N, y: M`), (PC: `M > 0 & N > 0`)
- Executing `x = x - y` yields (`m: M, n: N, x: N, y: 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`), (PC: `M > 0 & N > 0`)
- Executing `var x: Z = m` yields (`m: M, n: N, x: M`), (PC: `M > 0 & N > 0`)
- Executing `var y: Z = n` yields (`m: M, n: N, x: M, y: N`), (PC: `M > 0 & N > 0`)
- Executing `x = x + y` yields (`m: M, n: N, x: M + N, y: N`), (PC: `M > 0 & N > 0`)
- Executing `y = x - y` yields (`m: M, n: N, x: M + N, y: M`), (PC: `M > 0 & N > 0`)
- Executing `x = x - y` yields (`m: M, n: N, x: N, y: M`), (PC: `M > 0 & N > 0`)
- Executing `assert(x == n & y == m)` yields (`m: M, n: N, x: N, y: M`), (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`), (PC: `M > 0 & N > 0`)
- Executing `var x: Z = m` yields (`m: M, n: N, x: M`), (PC: `M > 0 & N > 0`)
- Executing `var y: Z = n` yields (`m: M, n: N, x: M, y: N`), (PC: `M > 0 & N > 0`)
- Executing `x = x + y` yields (`m: M, n: N, x: M + N, y: N`), (PC: `M > 0 & N > 0`)
- Executing `y = x - y` yields [illegible] `& N > 0`)
- Executing `x = x - y` yields [illegible] `> 0`)
- Executing `assert(x == n &` (`m: M, n: N, x: N, y: M`), (PC: `M > 0 & N > 0, N == N & M == M`)

The expression `M + N` cannot be further simplified at this stage because `M` and `N` are uninterpreted symbolic constants.

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