Software Correctness: The Construction of Correct Software

Contracts: Proof

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

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

AARHUS UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

A Proof From Linear Algebra

```
// linmap yields a * x + b for any a, x, and b
def linmap(a: Z, x: Z, b: Z): Z = {
  ...
}

// given (x - b) % a == 0 revmap yields (x - b) / a for any a, x, and b
def revmap(a: Z, x: Z, b: Z): Z = {
  ...
}

// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = ... // use function revmap
  return y
}
```

- The listing above shows three incompletely implemented functions `linmap`, `revmap`, and `compose`

A Proof From Linear Algebra

```
// linmap yields a * x + b for any a, x, and b
def linmap(a: Z, x: Z, b: Z): Z = {
  ...
}

// given (x - b) % a == 0 revmap yields (x - b) / a for any a, x, and b
def revmap(a: Z, x: Z, b: Z): Z = {
  ...
}

// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = ... // use function revmap
  return y
}
```

- The listing above shows three incompletely implemented functions `linmap`, `revmap`, and `compose`
- Suggested implementations are provided in the comments

A Proof From Linear Algebra

```
// linmap yields a * x + b for any a, x, and b
def linmap(a: Z, x: Z, b: Z): Z = {
  ...
}

// given (x - b) % a == 0 revmap yields (x - b) / a for any a, x, and b
def revmap(a: Z, x: Z, b: Z): Z = {
  ...
}

// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = ... // use function revmap
  return y
}
```

- The listing above shows three incompletely implemented functions `linmap`, `revmap`, and `compose`
- Suggested implementations are provided in the comments
- Let's implement the functions step by step

A Proof From Linear Algebra (linmap)

```
// linmap yields a * x + b for any a, x, and b
def linmap(a: Z, x: Z, b: Z): Z = {
  return a * x + b
}
```

- The implementation of `linmap` is easiest

A Proof From Linear Algebra (**linmap**)

```
// linmap yields a * x + b for any a, x, and b
def linmap(a: Z, x: Z, b: Z): Z = {
  return a * x + b
}
```

- The implementation of `linmap` is easiest
- We can simply copy the expression from the comment into a **`return`** statement

A Proof From Linear Algebra (**linmap**)

```
// linmap yields a * x + b for any a, x, and b
def linmap(a: Z, x: Z, b: Z): Z = {
  return a * x + b
}
```

- The implementation of `linmap` is easiest
- We can simply copy the expression from the comment into a **`return`** statement
- Let's leave it there for now

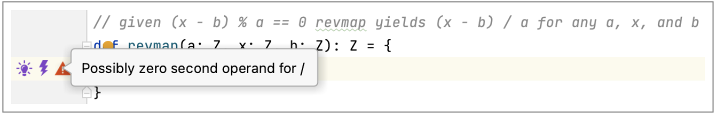
A Proof From Linear Algebra (**revmap**)

```
// given (x - b) % a == 0 revmap yields (x - b) / a for any a, x, and b
def revmap(a: Z, x: Z, b: Z): Z = {
  return (x - b) / a
}
```

- The implementation of `revmap` does not look challenging either

A Proof From Linear Algebra (`revmap`)

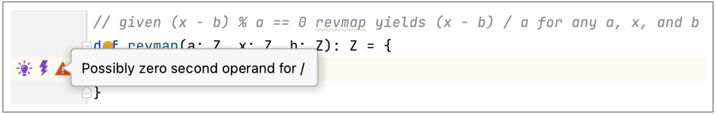
```
// given (x − b) % a == 0 revmap yields (x − b) / a for any a, x, and b
def revmap(a: Z, x: Z, b: Z): Z = {
  return (x − b) / a
}
```



- The implementation of `revmap` does not look challenging either
- Variable `a` referred to in the **`return`** statement might be zero

A Proof From Linear Algebra (**revmap**)

```
// given (x − b) % a == 0 revmap yields (x − b) / a for any a, x, and b
def revmap(a: Z, x: Z, b: Z): Z = {
  return (x − b) / a
}
```



- The implementation of `revmap` does not look challenging either
- Variable `a` referred to in the **`return`** statement might be zero
- We must add a requires clause to ensure the second operand of `/` is not zero

```
Contract(
  Requires(a != 0)
)
```

A Proof From Linear Algebra (**revmap**)

```
// given (x - b) % a == 0 revmap yields (x - b) / a for any a, x, and b
def revmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0)
  )
  return (x - b) / a
}
```

A Proof From Linear Algebra (**revmap**)

```
// given (x - b) % a == 0 revmap yields (x - b) / a for any a, x, and b
def revmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0)
  )
  return (x - b) / a
}
```

- In fact we should also add `(x - b) % a == 0` there as stated in the comment

A Proof From Linear Algebra (**revmap**)

```
// given (x - b) % a == 0 revmap yields (x - b) / a for any a, x, and b
def revmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0, (x - b) % a == 0)
  )
  return (x - b) / a
}
```

A Proof From Linear Algebra (**revmap**)

```
// given (x - b) % a == 0 revmap yields (x - b) / a for any a, x, and b
def revmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0, (x - b) % a == 0)
  )
  return (x - b) / a
}
```

- That’s it for now

A Proof From Linear Algebra (**revmap**)

```
// given (x − b) % a == 0 revmap yields (x − b) / a for any a, x, and b
def revmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0, (x − b) % a == 0)
  )
  return (x − b) / a
}
```

- That's it for now
- Let's turn to function `compose`

A Proof From Linear Algebra (**compose**)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = ... // use function revmap
  return y
}
```

A Proof From Linear Algebra (**compose**)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = ... // use function revmap
  return y
}
```

- Observe, `z == linmap(a, x, b) && y == revmap(a, z, b)`

A Proof From Linear Algebra (compose)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = ... // use function revmap
  return y
}
```

- Observe, `z == linmap(a, x, b) && y == revmap(a, z, b)` implies `z == a * x + b && y == (z - b) / a`

A Proof From Linear Algebra (**compose**)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = ... // use function revmap
  return y
}
```

- Observe, `z == linmap(a, x, b) && y == revmap(a, z, b)`
  implies `z == a * x + b && y == (z - b) / a`
  implies `y == (a * x + b - b) / a`

A Proof From Linear Algebra (**compose**)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = ... // use function revmap
  return y
}
```

- Observe, `z == linmap(a, x, b) && y == revmap(a, z, b)`
  implies `z == a * x + b && y == (z - b) / a`
  implies `y == (a * x + b - b) / a`
  implies `y == (a * x) / a`

A Proof From Linear Algebra (**compose**)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = ... // use function revmap
  return y
}
```

- Observe, `z == linmap(a, x, b) && y == revmap(a, z, b)`
  implies `z == a * x + b && y == (z - b) / a`
  implies `y == (a * x + b - b) / a`
  implies `y == (a * x) / a`
  implies `y == x`

A Proof From Linear Algebra (**compose**)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = ... // use function revmap
  return y
}
```

- Observe, `z == linmap(a, x, b) && y == revmap(a, z, b)`
  implies `z == a * x + b && y == (z - b) / a`
  implies `y == (a * x + b - b) / a`
  implies `y == (a * x) / a`
  implies `y == x`
- So, the missing function call is `revmap(a, y, b)`

A Proof From Linear Algebra (compose)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```

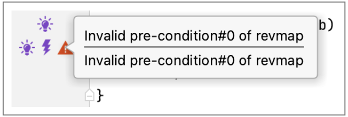
A Proof From Linear Algebra (compose)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```

- There's a problem with function `compose`

A Proof From Linear Algebra (compose)

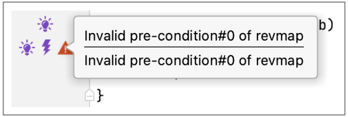
```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```



- There's a problem with function `compose`
- The pre-condition `a != 0` of function `revmap` is not met

A Proof From Linear Algebra (**compose**)

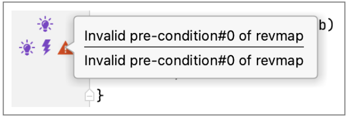
```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```



- There's a problem with function `compose`
- The pre-condition `a != 0` of function `revmap` is not met
- Because `a` is not modified in the function body,
  `a != 0` can only be enforced by a pre-condition

A Proof From Linear Algebra (compose)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  ...
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```



- There's a problem with function `compose`
- The pre-condition `a != 0` of function `revmap` is not met
- Because `a` is not modified in the function body, `a != 0` can only be enforced by a pre-condition
- We need to add a contract with the corresponding requires clause

A Proof From Linear Algebra (compose)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```

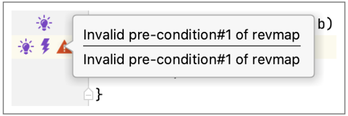
A Proof From Linear Algebra (**compose**)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```

- There's another problem with function `compose`!

A Proof From Linear Algebra (`compose`)

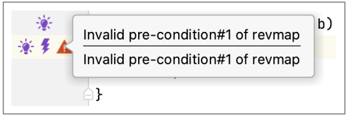
```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```



- There's another problem with function `compose`!
- The pre-condition `(At(y, 0) - b) % a == 0` of function `revmap` is not met
  (We've replaced `x` by `At(y, 0)` in the pre-condition according to the actual parameters.)

A Proof From Linear Algebra (**compose**)

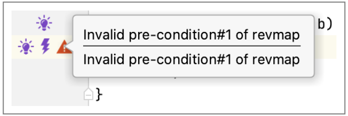
```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```



- There's another problem with function `compose`!
- The pre-condition `(At(y, 0) - b) % a == 0` of function `revmap` is not met (We've replaced `x` by `At(y, 0)` in the pre-condition according to the actual parameters.)
- Because `y` is assigned in `var y: Z = linmap(a, x, b)`, the pre-condition will have to established by the result of function `linmap`

A Proof From Linear Algebra (compose)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```



- There's another problem with function `compose`!
- The pre-condition `(At(y, 0) - b) % a == 0` of function `revmap` is not met
  (We've replaced `x` by `At(y, 0)` in the pre-condition according to the actual parameters.)
- Because `y` is assigned in `var y: Z = linmap(a, x, b)`,
  the pre-condition will have to established by the result of function `linmap`
- However, function `linmap` does not specify a post-condition

A Proof From Linear Algebra (**compose**)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```

```
def linmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Ensures(Res == a * x + b)
  )
  return a * x + b
}
```

- There's another problem with function `compose`!
- The pre-condition `(At(y, 0) - b) % a == 0` of function `revmap` is not met (We've replaced `x` by `At(y, 0)` in the pre-condition according to the actual parameters.)
- Because `y` is assigned in `var y: Z = linmap(a, x, b)`, the pre-condition will have to established by the result of function `linmap`
- However, function `linmap` does not specify a post-condition
- We need to add a contract with the corresponding ensures clause to `linmap`

A Proof From Linear Algebra (**compose**)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```

```
def linmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Ensures(Res == a * x + b)
  )
  return a * x + b
}
```

- There's another problem with function `compose`!
- The pre-condition `(At(y, 0) - b) % a == 0` of function `revmap` is not met (We've replaced `x` by `At(y, 0)` in the pre-condition according to the actual parameters.)
- Because `y` is assigned in `var y: Z = linmap(a, x, b)`, the pre-condition will have to established by the result of function `linmap`
- However, function `linmap` does not specify a post-condition
- We need to add a contract with the corresponding ensures clause to `linmap`
- The post-condition `At(y, 0) == a * x + b` implies the pre-condition `(At(y, 0) - b) % a == 0` of `revmap`

A Proof From Linear Algebra (**compose**)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```

```
def linmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Ensures(Res == a * x + b)
  )
  return a * x + b
}
```

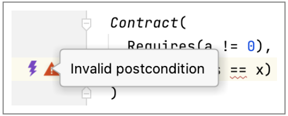
- There's another problem with function `compose`!
- The pre-condition `(At(y, 0) - b) % a == 0` of function `revmap` is not met (We've replaced `x` by `At(y, 0)` in the pre-condition according to the actual parameters.)
- Because `y` is assigned in `var y: Z = linmap(a, x, b)`, the pre-condition will have to established by the result of function `linmap`
- However, function `linmap` does not specify a post-condition
- We need to add a contract with the corresponding ensures clause to `linmap`
- The post-condition `At(y, 0) == a * x + b` implies the pre-condition `(At(y, 0) - b) % a == 0` of `revmap`
- Now let's add the post-condition `Res == x` to `compose`

A Proof From Linear Algebra (`compose`)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0),
    Ensures(Res == x)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```

A Proof From Linear Algebra (**compose**)

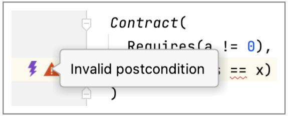
```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0),
    Ensures(Res == x)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```



- We can’t verify the function. The postcondition is invalid!

A Proof From Linear Algebra (**compose**)

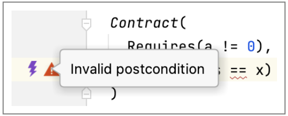
```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0),
    Ensures(Res == x)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```



- We can’t verify the function. The postcondition is invalid!
- We don’t have enough information to prove it

A Proof From Linear Algebra (**compose**)

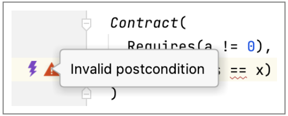
```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0),
    Ensures(Res == x)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```



- We can't verify the function. The postcondition is invalid!
- We don't have enough information to prove it
- We've reasoned informally above that the postcondition should be true

A Proof From Linear Algebra (compose)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0),
    Ensures(Res == x)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```



- We can't verify the function. The postcondition is invalid!
- We don't have enough information to prove it
- We've reasoned informally above that the postcondition should be true
- However, we've not specified a postcondition for `revmap` yet

A Proof From Linear Algebra (**compose**)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0),
    Ensures(Res == x)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```

```
def revmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0, (x - b) % a == 0),
    Ensures(Res == (x - b) / a)
  )
  return (x - b) / a
}
```

- We can't verify the function. The postcondition is invalid!
- We don't have enough information to prove it
- We've reasoned informally above that the postcondition should be true
- However, we've not specified a postcondition for `revmap` yet
- Let's do this

A Proof From Linear Algebra (compose)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0),
    Ensures(Res == x)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```

```
def revmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0, (x - b) % a == 0),
    Ensures(Res == (x - b) / a)
  )
  return (x - b) / a
}
```

- We can't verify the function. The postcondition is invalid!
- We don't have enough information to prove it
- We've reasoned informally above that the postcondition should be true
- However, we've not specified a postcondition for `revmap` yet
- Let's do this
- Now it's proved!

A Proof From Linear Algebra (**compose**)

```
// compose yields x for any a, x, and b
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0),
    Ensures(Res == x)
  )
  var y: Z = linmap(a, x, b)
  y = revmap(a, y, b)
  return y
}
```

```
def revmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0, (x - b) % a == 0),
    Ensures(Res == (x - b) / a)
  )
  return (x - b) / a
}
```

- We can't verify the function. The postcondition is invalid!
- We don't have enough information to prove it
- We've reasoned informally above that the postcondition should be true
- However, we've not specified a postcondition for `revmap` yet
- Let's do this
- Now it's proved!
- We can summarise and document our reasoning in Slang by providing deduce commands (relying on the contracts)

Logika Verified
Programming logic proof is accepted

A Proof From Linear Algebra (Summary)

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

// linmap yields a * x + b for any a, x, and b
def linmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Ensures(Res == a * x + b)
  )
  return a * x + b
}

// given (x - b) % a == 0
// revmap yields (x - b) / a for any a, x, and b
def revmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0, (x - b) % a == 0),
    Ensures(Res == (x - b) / a)
  )
  return (x - b) / a
}
```

```
// compose yields x for any a, x, and b
def compose1(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0),
    Ensures(Res == x)
  )
  var y: Z = linmap(a, x, b)
  Deduce(|- (y == a * x + b))
  y = revmap(a, y, b)
  Deduce(|- (a != 0))
  Deduce(|- ((a * x + b - b) % a == 0))
  Deduce(|- (At(y, 0) == a * x + b))
  Deduce(|- (y == ((At(y, 0) - b) / a)))
  Deduce(|- (y == ((a * x + b - b) / a)))
  return y
}
```

Exercise 1

Provide functions `linmap_spec`, `revmap_spec` and `compose_spec` completing the Slang program below where `x == compose_spec(a, x, b)` .

```
def linmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Ensures(Res == linmap_spec(a, x, b))
  )
  return a * x + b
}

def revmap(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0, (x - b) % a == 0),
    Ensures(Res == revmap_spec(a, x, b))
  )
  return (x - b) / a
}
```

```
def compose(a: Z, x: Z, b: Z): Z = {
  Contract(
    Requires(a != 0),
    Ensures(Res == compose_spec(a, x, b))
  )
  var y: Z = linmap(a, x, b)
  Deduce(|- (y == linmap_spec(a, x, b)))
  y = revmap(a, y, b)
  Deduce(|- (a != 0))
  Deduce(|- ((a * x + b - b) % a == 0))
  Deduce(|- (At(y, 0) == linmap_spec(a, x, b)))
  Deduce(|- (y == revmap_spec(a, At(y, 0), b)))
  Deduce(|- (y == revmap_spec(a, linmap_spec(a, x, b), b)))
  Deduce(|- (y == compose_spec(a, x, b)))
  return y
}
```

Exercise 2

Provide a function `inverse` with signature

`def inverse(a: Z, x: Z, b: Z)` (No return value!)

that ensures

```
revmap_spec(a, linmap_spec(a, x, b), b) == x
```

Aside. This function corresponds to a mathematical theorem in Slang

Slang Functions and Contracts

Slang Functions and Frames

Slang Functions as Facts

Slang Functions and Symbolic Execution

Summary

Example: Mutable Swapping with Frames

- Recall the mutable swapping program

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

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
Deduce(|- (x == n & y == m))
```

Example: Mutable Swapping with Frames

- Recall the mutable swapping program

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

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
Deduce(|- (x == n & y == m))
```

- We've replaced the final `assert` statement with a `Deduce` command

Example: Mutable Swapping with Frames

- Recall the mutable swapping program

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

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
Deduce(|- (x == n & y == m))
```

- We've replaced the final `assert` statement with a `Deduce` command
- Our intention is to **prove** this property of the swap program

Example: Mutable Swapping with Frames

- Recall the mutable swapping program

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

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
Deduce(|- (x == n & y == m))
```

- We've replaced the final `assert` statement with a `Deduce` command
- Our intention is to **prove** this property of the swap program
- Using what we've learned about programs and facts, we can express this without using variables `m` and `n`

Example: Mutable Swapping with Frames

- This simplifies the mutable swapping program

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

var x: Z = randomInt() // At(x, 0)

var y: Z = randomInt() // At(y, 0)

x = x + y
y = x - y
x = x - y
Deduce(|- (x == At(y, 0) & y == At(x, 0)))
```

Example: Mutable Swapping with Frames

- This simplifies the mutable swapping program

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

var x: Z = randomInt() // At(x, 0)

var y: Z = randomInt() // At(y, 0)

x = x + y
y = x - y
x = x - y
Deduce(|- (x == At(y, 0) & y == At(x, 0)))
```

- For the sake of this example let's restrict the values of the variables to positive integers

Example: Mutable Swapping with Frames

- Now, our example program looks as follows

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

var x: Z = randomInt() // At(x, 0)
assume(x > 0)
var y: Z = randomInt() // At(y, 0)
assume(y > 0)
x = x + y
y = x - y
x = x - y
Deduce(|- (x == At(y, 0) & y == At(x, 0)))
```

Example: Mutable Swapping with Frames

- Now, our example program looks as follows

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

var x: Z = randomInt() // At(x, 0)
assume(x > 0)
var y: Z = randomInt() // At(y, 0)
assume(y > 0)
x = x + y
y = x - y
x = x - y
Deduce(|- (x == At(y, 0) & y == At(x, 0)))
```

- With our example program in place, let's focus on the three assignments

Example: Mutable Swapping with Frames

- They contain different assignments to variables `x` and `y`

```
x = x + y
y = x - y
x = x - y
```

Example: Mutable Swapping with Frames

- They contain different assignments to variables `x` and `y`

```
x = x + y
y = x - y
x = x - y
```

- We're interested in the contracts governing these assignments

Example: Mutable Swapping with Frames

- They contain different assignments to variables `x` and `y`

```
x = x + y
y = x - y
x = x - y
```

- We're interested in the contracts governing these assignments
- Each of them
  - modifies a variable
  - has a post-condition
  - (and, possibly, a pre-condition depending on the expression on its right-hand side)

Example: Mutable Swapping with Frames

- They contain different assignments to variables `x` and `y`

```
x = x + y
y = x - y
x = x - y
```

- We're interested in the contracts governing these assignments
- Each of them
  - modifies a variable
  - has a post-condition
  - (and, possibly, a pre-condition depending on the expression on its right-hand side)
- let's consider one assignment after the other

Example: Mutable Swapping with Frames

```
x = x + y

y = x - y

x = x - y
```

Example: Mutable Swapping with Frames

```
// contract
//   modifies x
//   ensures x == At(x, 0) + y
x = x + y




y = x - y




x = x - y
```

- The first assignment modifies `x`
- The value `At(x, 0)` stems from the initial assignment

Example: Mutable Swapping with Frames

```
// contract
//   modifies x
//   ensures x == At(x, 0) + y
x = x + y

// contract
//   modifies y
//   ensures y == x - At(y, 0)
y = x - y




x = x - y
```

- The second assignment modifies `y`
- The value `At(y, 0)` stems from the initial assignment

Example: Mutable Swapping with Frames

```
// contract
//   modifies x
//   ensures x == At(x, 0) + y
x = x + y                          // At(x, 1)

// contract
//   modifies y
//   ensures y == x - At(y, 0)
y = x - y

// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

- The first assignment modifies `x`
- The value `At(x, 1)` stems from the indicated assignment

Statements and Contracts

- Not only assignments have contracts

Statements and Contracts

- Not only assignments have contracts
- Every Slang statement has a contract

Statements and Contracts

- Not only assignments have contracts
- Every Slang statement has a contract
- As if each statement was a function with a contract specification

Statements and Contracts

- Not only assignments have contracts
- Every Slang statement has a contract
- As if each statement was a function with a contract specification
- The contract reasoning for statements is built into Slang

Statements and Contracts

- Not only assignments have contracts
- Every Slang statement has a contract
- As if each statement was a function with a contract specification
- The contract reasoning for statements is built into Slang
- Let's make this explicit for the mutable swap program

Statements and Contracts

- Not only assignments have contracts
- Every Slang statement has a contract
- As if each statement was a function with a contract specification
- The contract reasoning for statements is built into Slang
- Let’s make this explicit for the mutable swap program
- We define a function for each assignment

Example: Mutable Swapping with Frames

```
// contract
//   modifies x
//   ensures x == At(x, 0) + y
x = x + y
```

```
// contract
//   modifies y
//   ensures y == x - At(y, 0)
y = x - y
```

```
// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

Example: Mutable Swapping with Frames

- In a function contract we cannot refer to old values such as `At(x, 0)`

```
// contract
//   modifies x
//   ensures x == At(x, 0) + y
x = x + y
```

```
// contract
//   modifies y
//   ensures y == x - At(y, 0)
y = x - y
```

```
// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

Example: Mutable Swapping with Frames

- In a function contract we cannot refer to old values such as `At(x, 0)`
- In a contract post-condition the old value is referred to as `In(x)`

```
// contract
//   modifies x
//   ensures x == At(x, 0) + y
x = x + y
```

```
// contract
//   modifies y
//   ensures y == x - At(y, 0)
y = x - y
```

```
// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

Example: Mutable Swapping with Frames

- In a function contract we cannot refer to old values such as `At(x, 0)`
- In a contract post-condition the old value is referred to as `In(x)`
- So, instead of writing `ensures x == At(x, 0) + y` we write `ensures x == In(x) + y`

```
// contract
//   modifies x
//   ensures x == At(x, 0) + y
x = x + y
```

```
// contract
//   modifies y
//   ensures y == x - At(y, 0)
y = x - y
```

```
// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

Example: Mutable Swapping with Frames

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}
```

```
// contract
//   modifies y
//   ensures y == x - At(y, 0)
y = x - y
```

```
// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

Example: Mutable Swapping with Frames

- We name the function for the first assignment `xplus`

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}
```

```
// contract
//   modifies y
//   ensures y == x - At(y, 0)
y = x - y
```

```
// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

Example: Mutable Swapping with Frames

- We name the function for the first assignment `xplus`
- It encapsulates the assignment `x = x + y` with a contract

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}
```

```
// contract
//   modifies y
//   ensures y == x - At(y, 0)
y = x - y
```

```
// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

Example: Mutable Swapping with Frames

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}

def yminus() {
  Contract(
    Modifies(y),
    Ensures(y == x - In(y))
  )
  y = x - y
}
```

```
// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

Example: Mutable Swapping with Frames

- We name the function for the second assignment `yminus`

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}

def yminus() {
  Contract(
    Modifies(y),
    Ensures(y == x - In(y))
  )
  y = x - y
}
```

```
// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

Example: Mutable Swapping with Frames

- We name the function for the second assignment `yminus`
- It encapsulates the assignment `y = x - y` with a contract

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}

def yminus() {
  Contract(
    Modifies(y),
    Ensures(y == x - In(y))
  )
  y = x - y
}
```

```
// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

Example: Mutable Swapping with Frames

- We name the function for the second assignment `yminus`
- It encapsulates the assignment `y = x - y` with a contract
- We treat this similar to the way we have dealt with the first assignment

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}

def yminus() {
  Contract(
    Modifies(y),
    Ensures(y == x - In(y))
  )
  y = x - y
}
```

```
// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

Example: Mutable Swapping with Frames

- We name the function for the second assignment `yminus`
- It encapsulates the assignment `y = x - y` with a contract
- We treat this similar to the way we have dealt with the first assignment
- The last assignment is a little different

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}

def yminus() {
  Contract(
    Modifies(y),
    Ensures(y == x - In(y))
  )
  y = x - y
}
```

```
// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

Example: Mutable Swapping with Frames

- We name the function for the second assignment `yminus`
- It encapsulates the assignment `y = x - y` with a contract
- We treat this similar to the way we have dealt with the first assignment
- The last assignment is a little different
- It refers to the “different” old value `At(x, 1)`

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}

def yminus() {
  Contract(
    Modifies(y),
    Ensures(y == x - In(y))
  )
  y = x - y
}
```

```
// contract
//   modifies x
//   ensures x == At(x, 1) - y)
x = x - y
```

Example: Mutable Swapping with Frames

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}

def yminus() {
  Contract(
    Modifies(y),
    Ensures(y == x - In(y))
  )
  y = x - y
}

def xminus(): Unit = {
  Contract(
    Modifies(x),
    Ensures(x == In(x) - y)
  )
  x = x - y
}
```

Example: Mutable Swapping with Frames

- The old value `At(x, 1)` must be provided by the context in which function `xminus` is called

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}

def yminus() {
  Contract(
    Modifies(y),
    Ensures(y == x - In(y))
  )
  y = x - y
}

def xminus(): Unit = {
  Contract(
    Modifies(x),
    Ensures(x == In(x) - y)
  )
  x = x - y
}
```

Example: Mutable Swapping with Frames

- The old value `At(x, 1)` must be provided by the context in which function `xminus` is called
- This was already the case `At(x, 0)` and `At(y, 0)`

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}

def yminus() {
  Contract(
    Modifies(y),
    Ensures(y == x - In(y))
  )
  y = x - y
}

def xminus(): Unit = {
  Contract(
    Modifies(x),
    Ensures(x == In(x) - y)
  )
  x = x - y
}
```

Example: Mutable Swapping with Frames

- The old value `At(x, 1)` must be provided by the context in which function `xminus` is called
- This was already the case `At(x, 0)` and `At(y, 0)`
- But now it becomes apparent
  that `In(x)` might refer to either depending on
  at which point in a program the function is called

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}

def yminus() {
  Contract(
    Modifies(y),
    Ensures(y == x - In(y))
  )
  y = x - y
}

def xminus(): Unit = {
  Contract(
    Modifies(x),
    Ensures(x == In(x) - y)
  )
  x = x - y
}
```

Example: Mutable Swapping with Frames

- The old value `At(x, 1)` must be provided by the context in which function `xminus` is called
- This was already the case `At(x, 0)` and `At(y, 0)`
- But now it becomes apparent that `In(x)` might refer to either depending on at which point in a program the function is called
- Instead of

```
x = x + y
y = x - y
x = x - y
```

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}

def yminus() {
  Contract(
    Modifies(y),
    Ensures(y == x - In(y))
  )
  y = x - y
}

def xminus(): Unit = {
  Contract(
    Modifies(x),
    Ensures(x == In(x) - y)
  )
  x = x - y
}
```

Example: Mutable Swapping with Frames

- The old value `At(x, 1)` must be provided by the context in which function `xminus` is called
- This was already the case `At(x, 0)` and `At(y, 0)`
- But now it becomes apparent that `In(x)` might refer to either depending on at which point in a program the function is called
- Instead of

```
x = x + y
y = x - y
x = x - y
```

we can write

```
xplus()
yminus()
xminus()
```

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}

def yminus() {
  Contract(
    Modifies(y),
    Ensures(y == x - In(y))
  )
  y = x - y
}

def xminus(): Unit = {
  Contract(
    Modifies(x),
    Ensures(x == In(x) - y)
  )
  x = x - y
}
```

Example: Mutable Swapping Function

- Suppose we have defined a mutable swapping function `swapA`

```
def swapA() {
  Contract(
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  x = x + y
  y = x - y
  x = x - y
}
```

Example: Mutable Swapping Function

- Suppose we have defined a mutable swapping function `swapA`

```
def swapA() {
  Contract(
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  x = x + y
  y = x - y
  x = x - y
}
```

- We can replace the three assignments by the newly defined functions

Example: Mutable Swapping Function

- We get the function swapB

```
def swapB() {
  Contract(
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  xplus()
  yminus()
  xminus()
}
```

- It has the same functionality as function swapA

Exercise 3

- Prove

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

var x: Z = randomInt() // At(x, 0)
assume(x > 0)
var y: Z = randomInt() // At(y, 0)
assume(y > 0)

...

swapA()
Deduce(|- (x == At(y, 0) & y == At(x, 0)))
```

where you insert the definition of `swapA` for `...`

- Prove that all intermediate values occurring in the body of function `swapA` are positive

Exercise 4

- Prove

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

var x: Z = randomInt() // At(x, 0)
assume(x > 0)
var y: Z = randomInt() // At(y, 0)
assume(y > 0)

...

swapB()
Deduce(|- (x == At(y, 0) & y == At(x, 0)))
```

where you insert the definition of `swapB` and the supporting functions for `...`

- Prove that all intermediate values occurring in the body of function `swapB` are positive

Example: Mutable Swapping Function (SwapA)

- Consider function swapA once more

```
def swapA() {
  Contract(
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  x = x + y
  y = x - y
  x = x - y
}
```

Example: Mutable Swapping Function (`SwapA`)

- Consider function `swapA` once more

```
def swapA() {
  Contract(
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  x = x + y
  y = x - y
  x = x - y
}
```

- The fact corresponding to the three assignments is just like what we've seen before

Example: Mutable Swapping Function (**SwapA**)

- Consider function `swapA` once more

```
def swapA() {
  Contract(
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  x = x + y
  y = x - y
  x = x - y
}
```

- The fact corresponding to the three assignments is just like what we've seen before
- We can look at it in Logika

Example: Mutable Swapping Function **`SwapA`** as Fact

- The fact for the function body of `swapA` is just as expected

```
def swap() {
  Contract(
    Requires(x > 0, y > 0),
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  x = x + y
  y = x - y
  x = x - y
}
```

```
{
  At(x, 0) > 0;
  At(y, 0) > 0;
  At(x, 1) == At(x, 0) + At(y, 0);
  y == At(x, 1) - At(y, 0);
  x == At(x, 1) - y;
  x == At(y, 0);
  y == At(x, 0)
}
```

Example: Mutable Swapping Function **`SwapA`** as Fact

- The fact for the function body of `swapA` is just as expected
- Identifying `At(x, 0)` with `In(x)` and `At(y, 0)` with `In(y)` it is easy to see how the post-condition is established

```
def swap() {
  Contract(
    Requires(x > 0, y > 0),
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  x = x + y
  y = x - y
  x = x - y
}
```

```
{
  At(x, 0) > 0;
  At(y, 0) > 0;
  At(x, 1) == At(x, 0) + At(y, 0);
  y == At(x, 1) - At(y, 0);
  x == At(x, 1) - y;
  x == At(y, 0);
  y == At(x, 0)
}
```

Example: Mutable Swapping Function `SwapA` as Fact

- The fact for the function body of `swapA` is just as expected
- Identifying `At(x, 0)` with `In(x)` and `At(y, 0)` with `In(y)` it is easy to see how the post-condition is established
- This provides a view from the inside of the function

```
def swap() {
  Contract(
    Requires(x > 0, y > 0),
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  x = x + y
  y = x - y
  x = x - y
}
```

```
{
  At(x, 0) > 0;
  At(y, 0) > 0;
  At(x, 1) == At(x, 0) + At(y, 0);
  y == At(x, 1) - At(y, 0);
  x == At(x, 1) - y;
  x == At(y, 0);
  y == At(x, 0)
}
```

Example: Mutable Swapping Function **`SwapA`** as Fact

- The fact for the function body of `swapA` is just as expected
- Identifying `At(x, 0)` with `In(x)` and `At(y, 0)` with `In(y)` it is easy to see how the post-condition is established
- This provides a view from the inside of the function
- From the outside it is seen in a function call to `swapA`

```
def swap() {
  Contract(
    Requires(x > 0, y > 0),
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  x = x + y
  y = x - y
  x = x - y
}
```

```
{
  At(x, 0) > 0;
  At(y, 0) > 0;
  At(x, 1) == At(x, 0) + At(y, 0);
  y == At(x, 1) - At(y, 0);
  x == At(x, 1) - y;
  x == At(y, 0);
  y == At(x, 0)
}
```

Example: Mutable Swapping Function **SwapA** as Fact

- From the outside only the contract of `swapA` is seen

```
swap()
Deduce(|- (x == At(y, 0) & y == At(x, 0)))
```

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

Example: Mutable Swapping Function **swapA** as Fact

- From the outside only the contract of `swapA` is seen
- The post-condition `x == In(y), y == In(x)` of `swapA` provides directly the facts needed to prove the deduction

```
swap()
Deduce(|- (x == At(y, 0) & y == At(x, 0)))
```

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

Example: Mutable Swapping Function `swapA` as Fact

- From the outside only the contract of `swapA` is seen
- The post-condition `x == In(y), y == In(x)` of `swapA` provides directly the facts needed to prove the deduction
- The modifies clause `Modifies(x, y)` specifies which variables need to be renamed using the `At`-notation

```
swap()
Deduce(|- (x == At(y, 0) & y == At(x, 0)))
```

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

Example: Mutable Swapping Function **SwapB**

- Consider function `swapB` with the function calls in the body

```
def swapB() {
  Contract(
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  xplus()
  yminus()
  xminus()
}
```

Example: Mutable Swapping Function **swapB**

- Consider function `swapB` with the function calls in the body

```
def swapB() {
  Contract(
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  xplus()
  yminus()
  xminus()
}
```

- The contracts for the three functions called in the body model the assignments closely

Example: Mutable Swapping Function **`swapB`** as Fact

- The fact for the function body of `swapB` is the same as `swapA`

```
def swap(): Unit = {
  Contract(
    Requires(x > 0, y > 0),
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  xplus()
  yminus()
  xminus()
}
```

```
{
  At(x, 0) > 0;
  At(y, 0) > 0;
  At(x, 1) == At(x, 0) + At(y, 0);
  y == At(x, 1) - At(y, 0);
  x == At(x, 1) - y;
  x == At(y, 0);
  y == At(x, 0)
}
```

Example: Mutable Swapping Function **SwapB** as Fact

- The fact for the function body of `swapB` is the same as `swapA`
- The contracts we've specified express the implicit contracts that govern assignments

```
def swap(): Unit = {
  Contract(
    Requires(x > 0, y > 0),
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  xplus()
  yminus()
  xminus()
}
```

```
{
  At(x, 0) > 0;
  At(y, 0) > 0;
  At(x, 1) == At(x, 0) + At(y, 0);
  y == At(x, 1) - At(y, 0);
  x == At(x, 1) - y;
  x == At(y, 0);
  y == At(x, 0)
}
```

Example: Mutable Swapping Function **SwapB** as Fact

- The fact for the function body of `swapB` is the same as `swapA`
- The contracts we've specified express the implicit contracts that govern assignments
- Seen from the outside by way of a call `swapA` and `swapB` are indistinguishable: they have identical contracts

```
def swap(): Unit = {
  Contract(
    Requires(x > 0, y > 0),
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  xplus()
  yminus()
  xminus()
}
```

```
{
  At(x, 0) > 0;
  At(y, 0) > 0;
  At(x, 1) == At(x, 0) + At(y, 0);
  y == At(x, 1) - At(y, 0);
  x == At(x, 1) - y;
  x == At(y, 0);
  y == At(x, 0)
}
```

Example: Mutable Swapping Function **SwapB** as Fact

- The fact for the function body of `swapB` is the same as `swapA`
- The contracts we've specified express the implicit contracts that govern assignments
- Seen from the outside by way of a call `swapA` and `swapB` are indistinguishable: they have identical contracts
- We can regard functions like theorems where the body is a proof.

```
def swap(): Unit = {
  Contract(
    Requires(x > 0, y > 0),
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  xplus()
  yminus()
  xminus()
}
```

```
{
  At(x, 0) > 0;
  At(y, 0) > 0;
  At(x, 1) == At(x, 0) + At(y, 0);
  y == At(x, 1) - At(y, 0);
  x == At(x, 1) - y;
  x == At(y, 0);
  y == At(x, 0)
}
```

Example: Mutable Swapping Function **`SwapB`** as Fact

- The fact for the function body of `swapB` is the same as `swapA`
- The contracts we've specified express the implicit contracts that govern assignments
- Seen from the outside by way of a call `swapA` and `swapB` are indistinguishable: they have identical contracts
- We can regard functions like theorems where the body is a proof.
- Using the theorem does not require knowledge of its proof

```
def swap(): Unit = {
  Contract(
    Requires(x > 0, y > 0),
    Modifies(x, y),
    Ensures(x == In(y), y == In(x))
  )
  xplus()
  yminus()
  xminus()
}
```

```
{
  At(x, 0) > 0;
  At(y, 0) > 0;
  At(x, 1) == At(x, 0) + At(y, 0);
  y == At(x, 1) - At(y, 0);
  x == At(x, 1) - y;
  x == At(y, 0);
  y == At(x, 0)
}
```

Slang Functions and Contracts

Slang Functions and Frames

Slang Functions as Facts

Slang Functions and Symbolic Execution

Summary

Problems

- Function calls pose several challenges concerning symbolic execution

Problems

- Function calls pose several challenges concerning symbolic execution
  (1) Function parameters introduce new temporary variables

Problems

- Function calls pose several challenges concerning symbolic execution
  (1) Function parameters introduce new temporary variables
  (2) The same parameter name may be used in different functions

Problems

- Function calls pose several challenges concerning symbolic execution
  (1) Function parameters introduce new temporary variables
  (2) The same parameter name may be used in different functions
  (3) Functions may call other functions

Problems

- Function calls pose several challenges concerning symbolic execution
  (1) Function parameters introduce new temporary variables
  (2) The same parameter name may be used in different functions
  (3) Functions may call other functions
  (4) Functions may contain recursive calls

Problems

- Function calls pose several challenges concerning symbolic execution
  (1) Function parameters introduce new temporary variables
  (2) The same parameter name may be used in different functions
  (3) Functions may call other functions
  (4) Functions may contain recursive calls
  (5) Functions may be nested

Problems

- Function calls pose several challenges concerning symbolic execution
  (1) Function parameters introduce new temporary variables
  (2) The same parameter name may be used in different functions
  (3) Functions may call other functions
  (4) Functions may contain recursive calls
  (5) Functions may be nested
  (6) Function calls may be nested

Problems

- Function calls pose several challenges concerning symbolic execution
  (1) Function parameters introduce new temporary variables
  (2) The same parameter name may be used in different functions
  (3) Functions may call other functions
  (4) Functions may contain recursive calls
  (5) Functions may be nested
  (6) Function calls may be nested
- In fact, some of these problem already appear when dealing with loops

Problems

- Function calls pose several challenges concerning symbolic execution
  (1) Function parameters introduce new temporary variables
  (2) The same parameter name may be used in different functions
  (3) Functions may call other functions
  (4) Functions may contain recursive calls
  (5) Functions may be nested
  (6) Function calls may be nested
- In fact, some of these problem already appear when dealing with loops
- We will deal with (1) and (2) disallowing (3) to (6) for now

Variable Names

- Consider function `shift` below

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}
```

Variable Names

- Consider function `shift` below

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}
```

- The function has the parameters `p`, `y` and `N`

Variable Names

- Consider function `shift` below

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}
```

- The function has the parameters `p`, `y` and `N`
- It refers to global variables `q`, `x`

Variable Names

- Consider function `shift` below

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}
```

- The function has the parameters `p`, `y` and `N`
- It refers to global variables `q`, `x`
- We need to rename `p`, `y` and `N`, the other two remain unchanged

Variable Names

- Consider function `shift` below

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}
```

- The function has the parameters `p`, `y` and `N`
- It refers to global variables `q`, `x`
- We need to rename `p`, `y` and `N`, the other two remain unchanged
- Let's prefix each of the three names with the name of the function `shift_`: `shift_p`, `shift_y` and `shift_N`

Variable Names

- Consider function `shift` below

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}
```

Variable Names

- Consider function `shift` below

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}
```

- as if it were

```
def shift(shift_p: Z, shift_y: Z, shift_N: Z) {
  Contract(
    Requires(x * shift_p + shift_y * q == shift_N),
    Modifies(x, q),
    Ensures(x * shift_p + shift_y * q == shift_N)
  )
  x = x - shift_y
  q = q + shift_p
}
```

Variable Names

- We symbolically execute the function

```
def shift(p: Z, y: Z, N: Z) {
  ...
}
```

Variable Names

- We symbolically execute the function

```
def shift(p: Z, y: Z, N: Z) {
  ...
}
```

- A call to the function now assigns values to the parameters

```
shift_p = ...
shift_y = ...
shift_N = ...
```

Variable Names

- We symbolically execute the function

```
def shift(p: Z, y: Z, N: Z) {
  ...
}
```

- A call to the function now assigns values to the parameters

```
shift_p = ...
shift_y = ...
shift_N = ...
```

- This approach does not generalise to arbitrary programs

Variable Names

- We symbolically execute the function

```
def shift(p: Z, y: Z, N: Z) {
  ...
}
```

- A call to the function now assigns values to the parameters

```
shift_p = ...
shift_y = ...
shift_N = ...
```

- This approach does not generalise to arbitrary programs
- Permitting (3) to (6) and (5) from slide (41) makes this method **unsound**

Variable Names

- We symbolically execute the function

```
def shift(p: Z, y: Z, N: Z) {
  ...
}
```

- A call to the function now assigns values to the parameters

```
shift_p = ...
shift_y = ...
shift_N = ...
```

- This approach does not generalise to arbitrary programs
- Permitting (3) to (6) and (5) from slide (41) makes this method **unsound**
- Being unsound means that the symbolic execution would not describe the program behaviour accurately

Variable Names

- We symbolically execute the function

```
def shift(p: Z, y: Z, N: Z) {
  ...
}
```

- A call to the function now assigns values to the parameters

```
shift_p = ...
shift_y = ...
shift_N = ...
```

- This approach does not generalise to arbitrary programs
- Permitting (3) to (6) and (5) from slide (41) makes this method **unsound**
- Being unsound means that the symbolic execution would not describe the program behaviour accurately
- We're interested in sound symbolic execution that permits us to make predictions about program behaviour

Initial Values of Global Variables

- Consider function `addy` below

```
def addy(y: Z) {
  Contract(
    Ensures(x == In(x) + y)
  )
  x = x + y
}
```

Initial Values of Global Variables

- Consider function `addy` below

```
def addy(y: Z) {
  Contract(
    Ensures(x == In(x) + y)
  )
  x = x + y
}
```

- To deal with the value `In(x)` we introduce an implicit parameter `addy_In(x)`

Initial Values of Global Variables

- Consider function `addy` below

```
def addy(y: Z) {
  Contract(
    Ensures(x == In(x) + y)
  )
  x = x + y
}
```

- To deal with the value `In(x)` we introduce an implicit parameter `addy_In(x)`
- The parameter `addy_In(x)` is assigned the value of variable `x` when the other parameters receive their value

```
addy_In(x) = x
```

Return Values

- Consider function `add` below

```
def add(x: Z, y: Z): Z = {
  Contract(
    Ensures(Res == x + y)
  )
  return x + y
}
```

Return Values

- Consider function add below

```
def add(x: Z, y: Z): Z = {
  Contract(
    Ensures(Res == x + y)
  )
  return x + y
}
```

- Because calls are not nested add may only occur in assignments

```
z = add(x, y)
```

Return Values

- Consider function `add` below

```
def add(x: Z, y: Z): Z = {
  Contract(
    Ensures(Res == x + y)
  )
  return x + y
}
```

- Because calls are not nested `add` may only occur in assignments

```
z = add(x, y)
```

- Symbolic execution of `return x + y` is then simply subsumed by the assignment to `z`

Symbolic Execution of the Program

- Using function `shift`

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}
```

Symbolic Execution of the Program

- Using function shift

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}
```

let's symbolically execute

```
assume(x + q == N)
shift(1, 1, N)
assert(x + q == N)
```

Symbolic Execution of the Program

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}

assume(x + q == N)
shift(1, 1, N)
assert(x + q == N)
```

Symbolic Execution of the Program

- (x: X, q: Q, N: NN),
  (PC: X + Q == NN)

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}

assume(x + q == N)
shift(1, 1, N)
assert(x + q == N)
```

Symbolic Execution of the Program

- (x: X, q: Q, N: NN),
  (PC: X + Q = NN)
- —

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}

assume(x + q == N)
shift(1, 1, N)
assert(x + q == N)
```

Symbolic Execution of the Program

- (`x: X`, `q: Q`, `N: NN`),
  (PC: `X + Q = NN`)
- —
- (`x: X`, `q: Q`, `N: NN`, `shift_p: 1`, `shift_y: 1`, `shift_N: NN`,
  `shift_In(x): X`, `shift_In(q): Q`),
  (PC: `X + Q = NN`)

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}

assume(x + q == N)
shift(1, 1, N)
assert(x + q == N)
```

Symbolic Execution of the Program

- (`x: X`, `q: Q`, `N: NN`),
  (PC: `X + Q = NN`)
- —
- (`x: X`, `q: Q`, `N: NN`, `shift_p: 1`, `shift_y: 1`, `shift_N: NN`,
  `shift_In(x): X`, `shift_In(q): Q`),
  (PC: `X + Q = NN`)
- (`x: X`, `q: Q`, `N: NN`, `...`),
  (PC: `X + Q = NN`, `X * 1 + 1 * Q = NN`)

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}

assume(x + q == N)
shift(1, 1, N)
assert(x + q == N)
```

Symbolic Execution of the Program

- (x: X, q: Q, N: NN),
  (PC: X + Q = NN)
- —
- (x: X, q: Q, N: NN, shift_p: 1, shift_y: 1, shift_N: NN,
  shift_In(x): X, shift_In(q): Q),
  (PC: X + Q = NN)
- (x: X, q: Q, N: NN, ...),
  (PC: X + Q = NN, X * 1 + 1 * Q = NN)
- (x: X − 1, q: Q, N: NN, ...),
  (PC: X + Q = NN, X * 1 + 1 * Q = NN)

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}

assume(x + q == N)
shift(1, 1, N)
assert(x + q == N)
```

Symbolic Execution of the Program

- (x: X, q: Q, N: NN),
  (PC: X + Q = NN)
- —
- (x: X, q: Q, N: NN, shift_p: 1, shift_y: 1, shift_N: NN,
  shift_In(x): X, shift_In(q): Q),
  (PC: X + Q = NN)
- (x: X, q: Q, N: NN, ...),
  (PC: X + Q = NN, X * 1 + 1 * Q = NN)
- (x: X - 1, q: Q, N: NN, ...),
  (PC: X + Q = NN, X * 1 + 1 * Q = NN)
- (x: X - 1, q: Q + 1, N: NN, ...),
  (PC: X + Q = NN, X * 1 + 1 * Q = NN)

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}

assume(x + q == N)
shift(1, 1, N)
assert(x + q == N)
```

Symbolic Execution of the Program

- (x: X, q: Q, N: NN),
  (PC: X + Q = NN)
- —
- (x: X, q: Q, N: NN, shift_p: 1, shift_y: 1, shift_N: NN,
  shift_In(x): X, shift_In(q): Q),
  (PC: X + Q = NN)
- (x: X, q: Q, N: NN, ...),
  (PC: X + Q = NN, X * 1 + 1 * Q = NN)
- (x: X - 1, q: Q, N: NN, ...),
  (PC: X + Q = NN, X * 1 + 1 * Q = NN)
- (x: X - 1, q: Q + 1, N: NN, ...),
  (PC: X + Q = NN, X * 1 + 1 * Q = NN)
- (x: X - 1, q: Q + 1, N: NN, ...),
  (PC: X + Q = NN, X * 1 + 1 * Q = NN,
  (X - 1) * 1 + 1 * (Q + 1) = NN)

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}

assume(x + q == N)
shift(1, 1, N)
assert(x + q == N)
```

Symbolic Execution of the Program

- `(x: X, q: Q, N: NN)`,
  `(PC: X + Q = NN)`
- —
- `(x: X, q: Q, N: NN, shift_p: 1, shift_y: 1, shift_N: NN,`
  `shift_In(x): X, shift_In(q): Q)`,
  `(PC: X + Q = NN)`
- `(x: X, q: Q, N: NN, ...)`,
  `(PC: X + Q = NN, X * 1 + 1 * Q = NN)`
- `(x: X - 1, q: Q, N: NN, ...)`,
  `(PC: X + Q = NN, X * 1 + 1 * Q = NN)`
- `(x: X - 1, q: Q + 1, N: NN, ...)`,
  `(PC: X + Q = NN, X * 1 + 1 * Q = NN)`
- `(x: X - 1, q: Q + 1, N: NN, ...)`,
  `(PC: X + Q = NN, X * 1 + 1 * Q = NN,`
  `(X - 1) * 1 + 1 * (Q + 1) = NN)`
- —

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}

assume(x + q == N)
shift(1, 1, N)
assert(x + q == N)
```

Symbolic Execution of the Program

- (`x: X`, `q: Q`, `N: NN`),
  (PC: `X + Q = NN`)
- —
- (`x: X`, `q: Q`, `N: NN`, `shift_p: 1`, `shift_y: 1`, `shift_N: NN`,
  `shift_In(x): X`, `shift_In(q): Q`),
  (PC: `X + Q = NN`)
- (`x: X`, `q: Q`, `N: NN`, `...`),
  (PC: `X + Q = NN`, `X * 1 + 1 * Q = NN`)
- (`x: X - 1`, `q: Q`, `N: NN`, `...`),
  (PC: `X + Q = NN`, `X * 1 + 1 * Q = NN`)
- (`x: X - 1`, `q: Q + 1`, `N: NN`, `...`),
  (PC: `X + Q = NN`, `X * 1 + 1 * Q = NN`)
- (`x: X - 1`, `q: Q + 1`, `N: NN`, `...`),
  (PC: `X + Q = NN`, `X * 1 + 1 * Q = NN`,
  `(X - 1) * 1 + 1 * (Q + 1) = NN`)
- —
- (`x: X - 1`, `q: Q + 1`, `N: NN`, `...`),
  (PC: `X + Q = M`, `X * 1 + 1 * Q = NN`,
  `(X - 1) * 1 + 1 * (Q + 1) = NN`,
  `(X - 1) + (Q + 1) = NN`)

```
def shift(p: Z, y: Z, N: Z) {
  Contract(
    Requires(x * p + y * q == N),
    Modifies(x, q),
    Ensures(x * p + y * q == N)
  )
  x = x - y
  q = q + p
}

assume(x + q == N)
shift(1, 1, N)
assert(x + q == N)
```

Exercise 5

Using

```
def xplus() {
  Contract(
    Modifies(x),
    Ensures(x == In(x) + y)
  )
  x = x + y
}

def yminus() {
  Contract(
    Modifies(y),
    Ensures(y == x - In(y))
  )
  y = x - y
}

def xminus(): Unit = {
  Contract(
    Modifies(x),
    Ensures(x == In(x) - y)
  )
  x = x - y
}
```

Symbolically execute

```
val x0: Z = x
val y0: Z = y
xplus()
yminus()
xminus()
assert(x == y0 & y = x0)
```

Summary

- We have looked at Slang functions in more detail
- We have focussed on the notion on contract and proof
- Considering assignments as a starting point we've analysed frames
- We've looked at symbolic execution of a simplified version of Slang