Software Correctness: The Construction of Correct Software

Loop Testing

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

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

Generating Test Cases from Implementations

Conditionals
- Conditionals as Facts
- Choosing Branches
- Symbolic Execution

Unfolded Iteration
- Bounded Iteration and Conditionals
- Termination

Unfolded Recursion
- Bounded Recursion and Conditionals
- Termination

Program Verification

Summary

Test Case Generation

- In the preceding lectures we have seen various ways
  - to trace facts through programs
  - to consider programs themselves as facts
  - to derive facts about executions paths of programs by symbolic execution
- All of these perspectives of programs can be exploited for proof and for testing
- Considering testing, we are particularly interested in obtaining test cases
- In the last lecture we have looked at iteration and recursion unfolding
- This technique permits us to look at testing of iteration and recursion as special cases on testing of conditionals
- To generate test cases we need contracts and programs

Generating Test Cases from Implementations

Conditionals
- Conditionals as Facts
- Choosing Branches
- Symbolic Execution

Unfolded Iteration
- Bounded Iteration and Conditionals
- Termination

Unfolded Recursion
- Bounded Recursion and Conditionals
- Termination

Program Verification

Summary

Example: Square Root Search

- Consider the following function for computing a step in a square root search

```
def sq_root_step() {
  Contract(
    Modifies(x, y)
  )
  val z: Z = (x + y) / 2
  if (z * z <= n) {
    x = z
  } else {
    y = z
  }
}
```

Example: Square Root Search

- Consider the following function for computing a step in a square root search

```
def sq_root_step() {
  Contract(
    Modifies(x, y)
  )
  val z: Z = (x + y) / 2
  if (z * z <= n) {
    x = z
  } else {
    y = z
  }
}
```

- Does the function preserve `x * x <= n`? That is, is it an invariant of the function body?

Example: Square Root Search

- We can specify the question in Slang

```
def sq_root_lb() {
  Contract(
    Requires(x * x <= n),
    Modifies(x, y),
    Ensures(x * x <= n)
  )
  sq_root_step()
}
```

Example: Square Root Search

- We can specify the question in Slang

```
def sq_root_lb() {
  Contract(
    Requires(x * x <= n),
    Modifies(x, y),
    Ensures(x * x <= n)
  )
  sq_root_step()
}
```

- We can use Logika's inter-procedural check to see whether this holds

Example: Square Root Search

- We can specify the question in Slang

```
def sq_root_lb() {
  Contract(
    Requires(x * x <= n),
    Modifies(x, y),
    Ensures(x * x <= n)
  )
  sq_root_step()
}
```

- We can use Logika's inter-procedural check to see whether this holds
- Let's have a look at the fact corresponding to function `sq_root_step`

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
At(x, 0) * At(x, 0) <= n                  & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2            & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))           & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)                  & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))          & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)                 & //    Assignment of z to y in else-branch
x * x <= n                                  // Post-condition from sq_root_lb
```

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
At(x, 0) * At(x, 0) <= n             & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2       & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))      & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)             & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))     & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)            & //    Assignment of z to y in else-branch
x * x <= n                             // Post-condition from sq_root_lb
```

- Recall the facts corresponding to conditionals of the shapes
  - C `=>` S_{fact}, where S is the program in the if-branch
  - `!`C `=>` T_{fact}, where T is the program in the else-branch

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
At(x, 0) * At(x, 0) <= n              & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2        & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))       & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)              & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))      & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)             & //    Assignment of z to y in else-branch
x * x <= n                              // Post-condition from sq_root_lb
```

- Recall the facts corresponding to conditionals of the shapes
  - C => S_{fact}, where S is the program in the if-branch
  - !C => T_{fact}, where T is the program in the else-branch
- If we want to test this program we have to choose specific branches

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
At(x, 0) * At(x, 0) <= n              & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2        & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))       & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)              & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))      & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)             & //    Assignment of z to y in else-branch
x * x <= n                              // Post-condition from sq_root_lb
```

- Recall the facts corresponding to conditionals of the shapes
  - C `=>` S_{fact}, where S is the program in the if-branch
  - `!`C `=>` T_{fact}, where T is the program in the else-branch
- If we want to test this program we have to choose specific branches
- For instance, `z * z <= n` to choose the if-branch

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
At(x, 0) * At(x, 0) <= n            & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2      & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))     & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)            & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))    & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)           & //    Assignment of z to y in else-branch
x * x <= n                            // Post-condition from sq_root_lb
```

- Recall the facts corresponding to conditionals of the shapes
  - C `=>` S_{fact}, where S is the program in the if-branch
  - `!`C `=>` T_{fact}, where T is the program in the else-branch
- If we want to test this program we have to choose specific branches
- For instance, `z * z <= n` to choose the if-branch
- We can conjoin this choice with the fact

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                                & // Choose if-branch
At(x, 0) * At(x, 0) <= n                  & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2            & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))           & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)                  & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))          & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)                 & //    Assignment of z to y in else-branch
x * x <= n                                  // Post-condition from sq_root_lb
```

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                              & // Choose if-branch
At(x, 0) * At(x, 0) <= n                & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2          & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))         & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)                & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))        & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)               & //    Assignment of z to y in else-branch
x * x <= n                                // Post-condition from sq_root_lb
```

- Those parts corresponding to the if-branch are selected by applying modus ponens

$$\frac{P \quad P \Rightarrow Q}{Q}$$

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                                    & // Choose if-branch
At(x, 0) * At(x, 0) <= n                      & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2                & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))               & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)                      & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))              & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)                     & //    Assignment of z to y in else-branch
x * x <= n                                      // Post-condition from sq_root_lb
```

- Those parts corresponding to the if-branch are selected by applying modus ponens

$$\frac{P \quad P \Rightarrow Q}{Q}$$

- Those parts corresponding to the if-branch are removed

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                                  & // Choose if-branch
At(x, 0) * At(x, 0) <= n                    & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2              & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))             & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)                    & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))            & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)                   & //    Assignment of z to y in else-branch
x * x <= n                                    // Post-condition from sq_root_lb
```

- Those parts corresponding to the if-branch are selected by applying modus ponens

$$\frac{P \quad P \Rightarrow Q}{Q}$$

- Those parts corresponding to the if-branch are removed
- They do not constrain any variable because `!(z * z <= n)` is false

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                                  & // Choose if-branch
At(x, 0) * At(x, 0) <= n                    & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2              & //  Assignment to z in sq_root_step
y == At(y, 0)                               & //  - Variable y unchanged in if-branch
x == z                                      & //    Assignment of z to x in if-branch
                                              //  - Variable x unchanged in else-branch
                                              //    Assignment of z to y in else-branch
x * x <= n                                    // Post-condition from sq_root_lb
```

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                                    & // Choose if-branch
At(x, 0) * At(x, 0) <= n                      & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2                & //  Assignment to z in sq_root_step
y == At(y, 0)                                 & //  - Variable y unchanged in if-branch
x == z                                        & //    Assignment of z to x in if-branch
                                                //  - Variable x unchanged in else-branch
                                                //    Assignment of z to y in else-branch
x * x <= n                                      // Post-condition from sq_root_lb
```

- Recall that `At(x, 0)` and `At(y, 0)` refer to the initial values of variables `x` and `y`

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                                  & // Choose if-branch
At(x, 0) * At(x, 0) <= n                    & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2              & //  Assignment to z in sq_root_step
y == At(y, 0)                               & //  - Variable y unchanged in if-branch
x == z                                      & //    Assignment of z to x in if-branch
                                              //  - Variable x unchanged in else-branch
                                              //    Assignment of z to y in else-branch
x * x <= n                                    // Post-condition from sq_root_lb
```

- Recall that `At(x, 0)` and `At(y, 0)` refer to the initial values of variables `x` and `y`
- Starting with
  input `x == 0`, `y == 0`, `n == 0`
  we should obtain the
  output `x == 0`, `y == 0`

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                                  & // Choose if-branch
At(x, 0) * At(x, 0) <= n                    & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2              & //  Assignment to z in sq_root_step
y == At(y, 0)                               & //  - Variable y unchanged in if-branch
x == z                                      & //    Assignment of z to x in if-branch
                                              //  - Variable x unchanged in else-branch
                                              //    Assignment of z to y in else-branch
x * x <= n                                    // Post-condition from sq_root_lb
```

- Recall that `At(x, 0)` and `At(y, 0)` refer to the initial values of variables `x` and `y`
- Starting with
  input `x == 0`, `y == 0`, `n == 0`
  we should obtain the
  output `x == 0`, `y == 0`
- If we added the post-condition `n < y * y`, this test case would no longer be valid

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                                &  // Choose if-branch
At(x, 0) * At(x, 0) <= n                  &  // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2            &  //  Assignment to z in sq_root_step
y == At(y, 0)                             &  //  - Variable y unchanged in if-branch
x == z                                    &  //    Assignment of z to x in if-branch
                                             //  - Variable x unchanged in else-branch
                                             //    Assignment of z to y in else-branch
x * x <= n                                &  // Post-condition from sq_root_lb
n < y * y                                    // Post-condition from sq_root_lb
```

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                              & // Choose if-branch
At(x, 0) * At(x, 0) <= n                & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2          & //  Assignment to z in sq_root_step
y == At(y, 0)                           & //  - Variable y unchanged in if-branch
x == z                                  & //    Assignment of z to x in if-branch
                                          //  - Variable x unchanged in else-branch
                                          //    Assignment of z to y in else-branch
x * x <= n                              & // Post-condition from sq_root_lb
n < y * y                                 // Post-condition from sq_root_lb
```

- Setting all variables to `0` is not true

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                                 & // Choose if-branch
At(x, 0) * At(x, 0) <= n                   & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2             & //  Assignment to z in sq_root_step
y == At(y, 0)                              & //  - Variable y unchanged in if-branch
x == z                                     & //    Assignment of z to x in if-branch
                                             //  - Variable x unchanged in else-branch
                                             //    Assignment of z to y in else-branch
x * x <= n                                 & // Post-condition from sq_root_lb
n < y * y                                    // Post-condition from sq_root_lb
```

- Setting all variables to `0` is not true
- However, negating `n < y * y` it becomes true

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                                 & // Choose if-branch
At(x, 0) * At(x, 0) <= n                   & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2             & //  Assignment to z in sq_root_step
y == At(y, 0)                              & //  - Variable y unchanged in if-branch
x == z                                     & //    Assignment of z to x in if-branch
                                             //  - Variable x unchanged in else-branch
                                             //    Assignment of z to y in else-branch
x * x <= n                                 & // Post-condition from sq_root_lb
n < y * y                                    // Post-condition from sq_root_lb
```

- Setting all variables to `0` is not true
- However, negating `n < y * y` it becomes true
- So, we've found a counterexample!

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                            & // Choose if-branch
At(x, 0) * At(x, 0) <= n              & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2        & //  Assignment to z in sq_root_step
y == At(y, 0)                         & //  - Variable y unchanged in if-branch
x == z                                & //    Assignment of z to x in if-branch
                                        //  - Variable x unchanged in else-branch
                                        //    Assignment of z to y in else-branch
x * x <= n                            & // Post-condition from sq_root_lb
n < y * y                               // Post-condition from sq_root_lb
```

- Setting all variables to `0` is not true
- However, negating `n < y * y` it becomes true
- So, we've found a counterexample!
- We would also have found it,
  had we used the test case just with `n < y * y` as post-condition

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
z * z <= n                                  & // Choose if-branch
At(x, 0) * At(x, 0) <= n                    & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2              & //  Assignment to z in sq_root_step
y == At(y, 0)                               & //  - Variable y unchanged in if-branch
x == z                                      & //    Assignment of z to x in if-branch
                                              //  - Variable x unchanged in else-branch
                                              //    Assignment of z to y in else-branch
x * x <= n                                  & // Post-condition from sq_root_lb
n < y * y                                     // Post-condition from sq_root_lb
```

- Setting all variables to `0` is not true
- However, negating `n < y * y` it becomes true
- So, we've found a counterexample!
- We would also have found it,
  had we used the test case just with `n < y * y` as post-condition
- This information can be fed into the original fact (before choosing the if-branch)

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
At(x, 0) * At(x, 0) <= n                 & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2           & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))          & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)                 & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))         & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)                & //    Assignment of z to y in else-branch
x * x <= n                               & // Post-condition from sq_root_lb
n < y * y                                  // Post-condition from sq_root_lb
```

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
At(x, 0) * At(x, 0) <= n                & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2          & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))         & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)                & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))        & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)               & //    Assignment of z to y in else-branch
x * x <= n                              & // Post-condition from sq_root_lb
n < y * y                                 // Post-condition from sq_root_lb
```

- We can mark those parts that are true

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
At(x, 0) * At(x, 0) <= n                 & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2           & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))          & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)                 & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))         & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)                & //    Assignment of z to y in else-branch
x * x <= n                               & // Post-condition from sq_root_lb
n < y * y                                  // Post-condition from sq_root_lb
```

- We can mark those parts that are true
- And those parts that are false

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
At(x, 0) * At(x, 0) <= n                    & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2              & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))             & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)                    & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))            & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)                   & //    Assignment of z to y in else-branch
x * x <= n                                  & // Post-condition from sq_root_lb
n < y * y                                     // Post-condition from sq_root_lb
```

- We can mark those parts that are true
- And those parts that are false
- Now we can trace back the fact `n < y * y` to the point where `y` was modified

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
At(x, 0) * At(x, 0) <= n             & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2       & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))      & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)             & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))     & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)            & //    Assignment of z to y in else-branch
x * x <= n                           & // Post-condition from sq_root_lb
n < y * y                              // Post-condition from sq_root_lb
```

- We can mark those parts that are true
- And those parts that are false
- Now we can trace back the fact `n < y * y` to the point where `y` was modified
- And discover the we need the fact `n < At(y, 0) * At(y, 0)` initially

Example: Square Root Search Fact

- The fact emanating from `sq_root_step` framed in the contract of `sq_root_lb`:

```
At(x, 0) * At(x, 0) <= n          & // Pre-condition from sq_root_lb
z == (At(x, 0) + At(y, 0)) / 2    & //  Assignment to z in sq_root_step
(z * z <= n) -> (y == At(y, 0))   & //  - Variable y unchanged in if-branch
(z * z <= n) -> (x == z)          & //    Assignment of z to x in if-branch
!(z * z <= n) -> (x == At(x, 0))  & //  - Variable x unchanged in else-branch
!(z * z <= n) -> (y == z)         & //    Assignment of z to y in else-branch
x * x <= n                        & // Post-condition from sq_root_lb
n < y * y                           // Post-condition from sq_root_lb
```

- We can mark those parts that are true
- And those parts that are false
- Now we can trace back the fact `n < y * y` to the point where `y` was modified
- And discover the we need the fact `n < At(y, 0) * At(y, 0)` initially
- In other words, we must add `n < y * y` as a pre-condition

Symbolic Execution of Square Root Search

```
def sq_root_step() {      // modifies x, y
  val z: Z = (x + y) / 2
  if (z * z <= n) {
    x = z
  } else {
    y = z
  }
} // return
```

```
def sq_root_lb_ub() {
  Contract( // modifies x, y
    Requires(x * x <= n),
    Ensures(x * x <= n, n < y * y)
  )
  sq_root_step()
}
```

Symbolic Execution of Square Root Search

```
def sq_root_step() {      // modifies x, y
  val z: Z = (x + y) / 2
  if (z * z <= n) {
    x = z
  } else {
    y = z
  }
} // return
```

```
def sq_root_lb_ub() {
  Contract( // modifies x, y
    Requires(x * x <= n),
    Ensures(x * x <= n, n < y * y)
  )
  sq_root_step()
}
```

- Executing `sq_root_lb_ub()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: **`true`**)

Symbolic Execution of Square Root Search

```
def sq_root_step() {      // modifies x, y
  val z: Z = (x + y) / 2
  if (z * z <= n) {
    x = z
  } else {
    y = z
  }
} // return
```

```
def sq_root_lb_ub() {
  Contract( // modifies x, y
    Requires(x * x <= n),
    Ensures(x * x <= n, n < y * y)
  )
  sq_root_step()
}
```

- Executing `sq_root_lb_ub()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: **`true`**)
- Executing `Requires(x * x <= n)` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `X0 * X0 <= N`)

Symbolic Execution of Square Root Search

```
def sq_root_step() { // modifies x, y
  val z: Z = (x + y) / 2
  if (z * z <= n) {
    x = z
  } else {
    y = z
  }
} // return
```

```
def sq_root_lb_ub() {
  Contract( // modifies x, y
    Requires(x * x <= n),
    Ensures(x * x <= n, n < y * y)
  )
  sq_root_step()
}
```

- Executing `sq_root_lb_ub()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `true`)
- Executing `Requires(x * x <= n)` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `X0 * X0 <= N`)
- Executing `sq_root_step()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `X0 * X0 <= N`)

Symbolic Execution of Square Root Search

```
def sq_root_step() {        // modifies x, y
  val z: Z = (x + y) / 2
  if (z * z <= n) {
    x = z
  } else {
    y = z
  }
} // return
```

```
def sq_root_lb_ub() {
  Contract( // modifies x, y
    Requires(x * x <= n),
    Ensures(x * x <= n, n < y * y)
  )
  sq_root_step()
}
```

- Executing `sq_root_lb_ub()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: **`true`**)
- Executing `Requires(x * x <= n)` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `X0 * X0 <= N`)
- Executing `sq_root_step()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `X0 * X0 <= N`)
- Executing `val z: Z = (x + y) / 2` yields (`x: X0`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`)

Symbolic Execution of Square Root Search

```
def sq_root_step() {      // modifies x, y
  val z: Z = (x + y) / 2
  if (z * z <= n) {
    x = z
  } else {
    y = z
  }
} // return
```

```
def sq_root_lb_ub() {
  Contract( // modifies x, y
    Requires(x * x <= n),
    Ensures(x * x <= n, n < y * y)
  )
  sq_root_step()
}
```

- Executing `sq_root_lb_ub()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `true`)
- Executing `Requires(x * x <= n)` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `X0 * X0 <= N`)
- Executing `sq_root_step()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `X0 * X0 <= N`)
- Executing `val z: Z = (x + y) / 2` yields (`x: X0`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`)
- Executing `if (z * z <= n) {` yields (`x: X0`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`, `Z * Z <= N`)

Symbolic Execution of Square Root Search

```
def sq_root_step() {        // modifies x, y
  val z: Z = (x + y) / 2
  if (z * z <= n) {
    x = z
  } else {
    y = z
  }
} // return
```

```
def sq_root_lb_ub() {
  Contract( // modifies x, y
    Requires(x * x <= n),
    Ensures(x * x <= n, n < y * y)
  )
  sq_root_step()
}
```

- Executing `sq_root_lb_ub()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `true`)
- Executing `Requires(x * x <= n)` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `X0 * X0 <= N`)
- Executing `sq_root_step()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `X0 * X0 <= N`)
- Executing `val z: Z = (x + y) / 2` yields (`x: X0`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`)
- Executing `if (z * z <= n) {` yields (`x: X0`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`, `Z * Z <= N`)
- Executing `x = z` yields (`x: Z`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`, `Z * Z <= N`)

Symbolic Execution of Square Root Search

```
def sq_root_step() {        // modifies x, y
  val z: Z = (x + y) / 2
  if (z * z <= n) {
    x = z
  } else {
    y = z
  }
} // return
```

```
def sq_root_lb_ub() {
  Contract( // modifies x, y
    Requires(x * x <= n),
    Ensures(x * x <= n, n < y * y)
  )
  sq_root_step()
}
```

- Executing `sq_root_lb_ub()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `true`)
- Executing `Requires(x * x <= n)` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `X0 * X0 <= N`)
- Executing `sq_root_step()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `X0 * X0 <= N`)
- Executing `val z: Z = (x + y) / 2` yields (`x: X0`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`)
- Executing `if (z * z <= n) {` yields (`x: X0`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`, `Z * Z <= N`)
- Executing `x = z` yields (`x: Z`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`, `Z * Z <= N`)
- Executing `} // return` yields (`x: Z`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`, `Z * Z <= N`)

Symbolic Execution of Square Root Search

```
def sq_root_step() {      // modifies x, y
  val z: Z = (x + y) / 2
  if (z * z <= n) {
    x = z
  } else {
    y = z
  }
} // return
```

```
def sq_root_lb_ub() {
  Contract( // modifies x, y
    Requires(x * x <= n),
    Ensures(x * x <= n, n < y * y)
  )
  sq_root_step()
}
```

- Executing `sq_root_lb_ub()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: **true**)
- Executing `Requires(x * x <= n)` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `X0 * X0 <= N`)
- Executing `sq_root_step()` yields (`x: X0`, `y: Y0`, `n: N`), (PC: `X0 * X0 <= N`)
- Executing `val z: Z = (x + y) / 2` yields (`x: X0`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`)
- Executing `if (z * z <= n) {` yields
  (`x: X0`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`, `Z * Z <= N`)
- Executing `x = z` yields
  (`x: Z`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`, `Z * Z <= N`)
- Executing `} // return` yields
  (`x: Z`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`, `Z * Z <= N`)
- Executing `Ensures(x * x <= n, n < y * y)` yields
  (`x: Z`, `y: Y0`, `n: N`, `z: Z`), (PC: `X0 * X0 <= N`, `Z == (X0 + Y0) / 2`, `Z * Z <= N`, `N < Y0 * Y0`)

Example: Iterative Square Root

- We can implement the computation of an integer square root as a binary search

```
@pure def sq_root(n: Z): Z = {
  var x: Z = 0
  var y: Z = n + 1
  while (x + 1 != y) {
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      x = z
    } else {
      y = z
    }
  }
  return x
}
```

Example: Iterative Square Root

- We can implement the computation of an integer square root as a binary search

```
@pure def sq_root(n: Z): Z = {
  var x: Z = 0
  var y: Z = n + 1
  while (x + 1 != y) {
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      x = z
    } else {
      y = z
    }
  }
  return x
}
```

- We can use symbolic execution or unfolding for test case generation

Example: Iterative Square Root

- Unfolded while-loop:

```
@pure def sq_root(n: Z): Z = {
  var x: Z = 0
  var y: Z = n + 1
  if (x + 1 != y) {
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      x = z
    } else {
      y = z
    }
    if (x + 1 != y) {
      val z: Z = (x + y) / 2
      if (z * z <= n) {
        x = z
      } else {
        y = z
      }
      while (x + 1 != y) {
        val z: Z = (x + y) / 2
        if (z * z <= n) {
          x = z
        } else {
          y = z
        }
      }
    }
  }
  return x
}
```

Example: Iterative Square Root

- Unfolded while-loop:

```
@pure def sq_root(n: Z): Z = {
  var x: Z = 0
  var y: Z = n + 1
  if (x + 1 != y) {
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      x = z
    } else {
      y = z
    }
    if (x + 1 != y) {
      val z: Z = (x + y) / 2
      if (z * z <= n) {
        x = z
      } else {
        y = z
      }
      while (x + 1 != y) {
        val z: Z = (x + y) / 2
        if (z * z <= n) {
          x = z
        } else {
          y = z
        }
      }
    }
  }
  return x
}
```

- This function is equivalent to the original one

Example: Iterative Square Root

- We don't want the final while-loop to be executed. Let's make this more precise

```
@pure def sq_root(n: Z): Z = {
  var x: Z = 0
  var y: Z = n + 1
  if (x + 1 != y) {
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      x = z
    } else {
      y = z
    }
    if (x + 1 != y) {
      val z: Z = (x + y) / 2
      if (z * z <= n) {
        x = z
      } else {
        y = z
      }
      if (x + 1 != y) {
        abort() // This location must never be reached
      }
    }
  }
  return x
}
```

Example: Iterative Square Root

- We don't want the final while-loop to be executed. Let's make this more precise

```
@pure def sq_root(n: Z): Z = {
  var x: Z = 0
  var y: Z = n + 1
  if (x + 1 != y) {
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      x = z
    } else {
      y = z
    }
    if (x + 1 != y) {
      val z: Z = (x + y) / 2
      if (z * z <= n) {
        x = z
      } else {
        y = z
      }
      if (x + 1 != y) {
        abort() // This location must never be reached
      }
    }
  }
  return x
}
```

- We use `abort()` to mark branches that we do not wish to consider

Example: Iterative Square Root

- We don't want the final while-loop to be executed. Let's make this more precise

```
@pure def sq_root(n: Z): Z = {
  var x: Z = 0
  var y: Z = n + 1
  if (x + 1 != y) {
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      x = z
    } else {
      y = z
    }
    if (x + 1 != y) {
      val z: Z = (x + y) / 2
      if (z * z <= n) {
        x = z
      } else {
        y = z
      }
      if (x + 1 != y) {
        abort() // This location must never be reached
      }
    }
  }
  return x
}
```

- We use `abort()` to mark branches that we do not wish to consider
- This function is **not** equivalent to the original one

Example: Iterative Square Root

- We don't want the final while-loop to be executed. Let's make this more precise

```
@pure def sq_root(n: Z): Z = {
  var x: Z = 0
  var y: Z = n + 1
  if (x + 1 != y) {
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      x = z
    } else {
      y = z
    }
    if (x + 1 != y) {
      val z: Z = (x + y) / 2
      if (z * z <= n) {
        x = z
      } else {
        y = z
      }
      if (x + 1 != y) {
        abort() // This location must never be reached
      }
    }
  }
  return x
}
```

- We use `abort()` to mark branches that we do not wish to consider
- This function is **not** equivalent to the original one
- Using it we set a bound on the iteration when analysing the while-loop

Example: Iterative Square Root

- This becomes difficult to handle “manually”

Example: Iterative Square Root

- This becomes difficult to handle “manually”
- We can use Logika to inspect the formulas at different iteration depths

Example: Iterative Square Root

- This becomes difficult to handle “manually”
- We can use Logika to inspect the formulas at different iteration depths
- With an increasing number of iterations the formulas grow

Example: Iterative Square Root

- This becomes difficult to handle “manually”
- We can use Logika to inspect the formulas at different iteration depths
- With an increasing number of iterations the formulas grow
- The facts become complex

Example: Iterative Square Root

- This becomes difficult to handle "manually"
- We can use Logika to inspect the formulas at different iteration depths
- With an increasing number of iterations the formulas grow
- The facts become complex
- We still can understand and analyse them when we find errors

Example: Iterative Square Root

- This becomes difficult to handle “manually”
- We can use Logika to inspect the formulas at different iteration depths
- With an increasing number of iterations the formulas grow
- The facts become complex
- We still can understand and analyse them when we find errors
- But we must rely on Logika to generate and manage those facts

Example: Iterative Square Root

- This becomes difficult to handle “manually”
- We can use Logika to inspect the formulas at different iteration depths
- With an increasing number of iterations the formulas grow
- The facts become complex
- We still can understand and analyse them when we find errors
- But we must rely on Logika to generate and manage those facts
- Let’s look at a few iterations

Example: Iterative Square Root Facts

```
At[Z]("sq_root_lb.n", 0) >= 0;
At(n, 0) == At[Z]("sq_root_lb.n", 0);
x == 0;
y == At(n, 0) + 1;
!(x + 1 != y);
At(Res, 0) == x;
n == At[Z]("sq_root_lb.n", 0);
y == n + 1
```

Example: Iterative Square Root Facts

```
At[Z]("sq_root_lb.n", 0) >= 0;
At(n, 0) == At[Z]("sq_root_lb.n", 0);
At(x, 0) == 0;
At(y, 0) == At(n, 0) + 1;
At(x, 0) + 1 != At(y, 0);
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (y == At(y, 0));
(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (x == At(z, 0));
!(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (x == At(x, 0));
!(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (y == At(z, 0));
!(x + 1 != y);
At(Res, 0) == x;
n == At[Z]("sq_root_lb.n", 0);
x == 0;
y == n + 1
```

Example: Iterative Square Root Facts

```
At[Z]("sq_root_lb.n", 0) >= 0;
At(n, 0) == At[Z]("sq_root_lb.n", 0);
At(x, 0) == 0;
At(y, 0) == At(n, 0) + 1;
At(x, 0) + 1 != At(y, 0);
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(x, 1) == At(z, 0));
!(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(y, 1) == At(z, 0));
At(x, 1) + 1 != At(y, 1);
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(z, 0) * At(z, 0) <= At(n, 0)) ->: (y == At(y, 0));
(At(z, 1) * At(z, 1) <= At(n, 0)) ->: !(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(x, 1) == At(x, 0));
(At(z, 1) * At(z, 1) <= At(n, 0)) ->: !(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (y == At(y, 1));
(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (y == At(y, 1));
(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (x == At(z, 1));
!(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(y, 1) == At(y, 0));
!(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(z, 0) * At(z, 0) <= At(n, 0)) ->: (x == At(x, 1));
!(At(z, 1) * At(z, 1) <= At(n, 0)) ->: !(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (x == At(x, 0));
!(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (x == At(x, 1));
!(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (y == At(z, 1));
!(x + 1 != y);
At(Res, 0) == x;
n == At[Z]("sq_root_lb.n", 0);
x == 0;
y == n + 1
```

Example: Iterative Square Root Facts

```
At[Z]("sq_root_lb.n", 0) >= 0;
At(n, 0) == At[Z]("sq_root_lb.n", 0);
At(x, 0) == 0;
At(y, 0) == At(n, 0) + 1;
At(x, 0) + 1 != At(y, 0);
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(x, 1) == At(z, 0));
!(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(y, 1) == At(z, 0));
At(x, 1) + 1 != At(y, 1);
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
(At(z, 1) * At(z, 1) <= At(n, 0)) ->: !(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(x, 1) == At(x, 0));
(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(x, 2) == At(z, 1));
!(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(y, 1) == At(y, 0));
!(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(y, 2) == At(z, 1));
At(x, 2) + 1 != At(y, 2);
At(z, 2) == (At(x, 2) + At(y, 2)) / 2;
(At(z, 2) * At(z, 2) <= At(n, 0)) ->: (At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(z, 0) * At(z, 0) <= At(n, 0)) ->: (y == At(y, 0));
(At(z, 2) * At(z, 2) <= At(n, 0)) ->: (At(z, 1) * At(z, 1) <= At(n, 0)) ->: !(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (y == At(y, 1));
(At(z, 2) * At(z, 2) <= At(n, 0)) ->: (At(z, 1) * At(z, 1) <= At(n, 0)) ->: (y == At(y, 1));
(At(z, 2) * At(z, 2) <= At(n, 0)) ->: !(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(x, 2) == At(x, 1));
(At(z, 2) * At(z, 2) <= At(n, 0)) ->: !(At(z, 1) * At(z, 1) <= At(n, 0)) ->: !(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(x, 2) == At(x, 0));
(At(z, 2) * At(z, 2) <= At(n, 0)) ->: !(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(x, 2) == At(x, 1));
(At(z, 2) * At(z, 2) <= At(n, 0)) ->: !(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (y == At(y, 2));
(At(z, 2) * At(z, 2) <= At(n, 0)) ->: (y == At(y, 2));
(At(z, 2) * At(z, 2) <= At(n, 0)) ->: (x == At(z, 2));
!(At(z, 2) * At(z, 2) <= At(n, 0)) ->: (At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(y, 2) == At(y, 0));
!(At(z, 2) * At(z, 2) <= At(n, 0)) ->: (At(z, 1) * At(z, 1) <= At(n, 0)) ->: !(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(y, 2) == At(y, 1));
!(At(z, 2) * At(z, 2) <= At(n, 0)) ->: (At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(y, 2) == At(y, 1));
!(At(z, 2) * At(z, 2) <= At(n, 0)) ->: (At(z, 1) * At(z, 1) <= At(n, 0)) ->: (x == At(x, 2));
!(At(z, 2) * At(z, 2) <= At(n, 0)) ->: !(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(z, 0) * At(z, 0) <= At(n, 0)) ->: (x == At(x, 1));
!(At(z, 2) * At(z, 2) <= At(n, 0)) ->: !(At(z, 1) * At(z, 1) <= At(n, 0)) ->: !(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (x == At(x, 0));
!(At(z, 2) * At(z, 2) <= At(n, 0)) ->: !(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (x == At(x, 1));
!(At(z, 2) * At(z, 2) <= At(n, 0)) ->: (x == At(x, 2));
!(At(z, 2) * At(z, 2) <= At(n, 0)) ->: (y == At(z, 2));
!(x + 1 != y);
At(Res, 0) == x;
n == At[Z]("sq_root_lb.n", 0);
x == 0;
y == n + 1
```

Termination of the Iterative Square Root

- Before, we have seen how to specify that a program terminates by way of a measure

Termination of the Iterative Square Root

- Before, we have seen how to specify that a program terminates by way of a measure
- The body of the loop `decreases` the measure at each iteration

Termination of the Iterative Square Root

- Before, we have seen how to specify that a program terminates by way of a measure
- The body of the loop `decreases` the measure at each iteration
- While the measure is bounded below by 0

Termination of the Iterative Square Root

- Before, we have seen how to specify that a program terminates by way of a measure
- The body of the loop `decreases` the measure at each iteration
- While the measure is bounded below by 0
- We can specify these properties of the measure using assertions

```
@pure def sq_root(n: Z): Z = {
  var x: Z = 0
  var y: Z = n + 1
  while (x + 1 != y) {
    val measure_yx_pre = y - x
    assert(measure_yx_pre >= 0)
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      x = z
    } else {
      y = z
    }
    val measure_yx_post = y - x
    assert(measure_yx_post < measure_yx_pre)
  }
  return x
}
```

Termination of the Iterative Square Root

- Before, we have seen how to specify that a program terminates by way of a measure
- The body of the loop `decreases` the measure at each iteration
- While the measure is bounded below by 0
- We can specify these properties of the measure using assertions

```
@pure def sq_root(n: Z): Z = {
  var x: Z = 0
  var y: Z = n + 1
  while (x + 1 != y) {
    val measure_yx_pre = y - x
    assert(measure_yx_pre >= 0)
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      x = z
    } else {
      y = z
    }
    val measure_yx_post = y - x
    assert(measure_yx_post < measure_yx_pre)
  }
  return x
}
```

- Of course, we can unfold this function, too

Termination of the Iterative Square Root

- Before, we have seen how to specify that a program terminates by way of a measure
- The body of the loop `decreases` the measure at each iteration
- While the measure is bounded below by 0
- We can specify these properties of the measure using assertions

```
@pure def sq_root(n: Z): Z = {
  var x: Z = 0
  var y: Z = n + 1
  while (x + 1 != y) {
    val measure_yx_pre = y - x
    assert(measure_yx_pre >= 0)
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      x = z
    } else {
      y = z
    }
    val measure_yx_post = y - x
    assert(measure_yx_post < measure_yx_pre)
  }
  return x
}
```

- Of course, we can unfold this function, too
- The `assert` statements are now considered in the fact of the unfolded function

Example: Iterative Square Root Measure Fact

```
At[Z]("sq_root_lb_term.n", 0) >= 0;
At(n, 0) == At[Z]("sq_root_lb_term.n", 0);
At(x, 0) == 0;
At(y, 0) == At(n, 0) + 1;
At(x, 0) + 1 != At(y, 0);
At(measure_yx_pre, 0) == At(y, 0) - At(x, 0);
At(measure_yx_pre, 0) >= 0;
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (y == At(y, 0));
(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (x == At(z, 0));
!(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (x == At(x, 0));
!(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (y == At(z, 0));
At(measure_yx_post, 0) == y - x;
At(measure_yx_post, 0) < At(measure_yx_pre, 0);
!(x + 1 != y);
At(Res, 0) == x;
n == At[Z]("sq_root_lb_term.n", 0);
x == 0;
y == n + 1
```

Example: Iterative Square Root Measure Fact

```
At[Z]("sq_root_lb_term.n", 0) >= 0;
At(n, 0) == At[Z]("sq_root_lb_term.n", 0);
At(x, 0) == 0;
At(y, 0) == At(n, 0) + 1;
At(x, 0) + 1 != At(y, 0);
At(measure_yx_pre, 0) == At(y, 0) - At(x, 0);
At(measure_yx_pre, 0) >= 0;
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(x, 1) == At(z, 0));
!(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(y, 1) == At(z, 0));
At(measure_yx_post, 0) == At(y, 1) - At(x, 1);
At(measure_yx_post, 0) < At(measure_yx_pre, 0);
At(x, 1) + 1 != At(y, 1);
At(measure_yx_pre, 1) == At(y, 1) - At(x, 1);
At(measure_yx_pre, 1) >= 0;
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(z, 0) * At(z, 0) <= At(n, 0)) ->: (y == At(y, 0));
(At(z, 1) * At(z, 1) <= At(n, 0)) ->: !(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(x, 1) == At(x, 0));
(At(z, 1) * At(z, 1) <= At(n, 0)) ->: !(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (y == At(y, 1));
(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (y == At(y, 1));
(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (x == At(z, 1));
!(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(z, 0) * At(z, 0) <= At(n, 0)) ->: (At(y, 1) == At(y, 0));
!(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (At(z, 0) * At(z, 0) <= At(n, 0)) ->: (x == At(x, 1));
!(At(z, 1) * At(z, 1) <= At(n, 0)) ->: !(At(z, 0) * At(z, 0) <= At(n, 0)) ->: (x == At(x, 0));
!(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (x == At(x, 1));
!(At(z, 1) * At(z, 1) <= At(n, 0)) ->: (y == At(z, 1));
At(measure_yx_post, 1) == y - x;
At(measure_yx_post, 1) < At(measure_yx_pre, 1);
!(x + 1 != y);
At(Res, 0) == x;
n == At[Z]("sq_root_lb_term.n", 0);
x == 0;
y == n + 1
```

Example: Recursive Square Root

- In the recursive version the local variables `x` and `y` become accumulator arguments

```
@pure def sq_root_rec(n: Z, x: Z, y: Z): Z = {
  if (x + 1 == y) {
    return x
  } else {
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      return sq_root_rec(n, z, y)
    } else {
      return sq_root_rec(n, x, z)
    }
  }
}

@pure def sq_root(n: Z): Z = {
  return sq_root_rec(n, 0, n+1)
}
```

Example: Recursive Square Root

- In the recursive version the local variables `x` and `y` become accumulator arguments

```
@pure def sq_root_rec(n: Z, x: Z, y: Z): Z = {
  if (x + 1 == y) {
    return x
  } else {
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      return sq_root_rec(n, z, y)
    } else {
      return sq_root_rec(n, x, z)
    }
  }
}

@pure def sq_root(n: Z): Z = {
  return sq_root_rec(n, 0, n+1)
}
```

- Let's look at the fact of the unfolded recursive square root function

Example: Recursive Square Root Facts

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) == At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) == At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
At(z, 0) * At(z, 0) <= At(n, 0);
n == At(n, 0);
x == At(z, 0);
y == At(y, 0)
```

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) == At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) == At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
n == At(n, 0);
x == At(x, 0);
y == At(z, 0)
```

Example: Recursive Square Root Facts

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) ==
  At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) ==
  At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
At(n, 1) == At(n, 0);
At(x, 1) == At(x, 0);
At(y, 1) == At(z, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
At(z, 1) * At(z, 1) <= At(n, 1);
n == At(n, 1);
x == At(z, 1);
y == At(y, 1)
```

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) ==
  At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) ==
  At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
At(z, 0) * At(z, 0) <= At(n, 0);
At(n, 1) == At(n, 0);
At(x, 1) == At(z, 0);
At(y, 1) == At(y, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
!(At(z, 1) * At(z, 1) <= At(n, 1));
n == At(n, 1);
x == At(x, 1);
y == At(z, 1)
```

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) ==
  At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) ==
  At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
At(n, 1) == At(n, 0);
At(x, 1) == At(x, 0);
At(y, 1) == At(z, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
!(At(z, 1) * At(z, 1) <= At(n, 1));
n == At(n, 1);
x == At(x, 1);
y == At(z, 1)
```

Example: Recursive Square Root Facts

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) ==
  At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) ==
  At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
At(n, 1) == At(n, 0);
At(x, 1) == At(x, 0);
At(y, 1) == At(z, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
At(z, 1) * At(z, 1) <= At(n, 1);
n == At(n, 1);
x == At(z, 1);
y == At(y, 1)
```

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) ==
  At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) ==
  At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
At(z, 0) * At(z, 0) <= At(n, 0);
At(n, 1) == At(n, 0);
At(x, 1) == At(z, 0);
At(y, 1) == At(y, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
!(At(z, 1) * At(z, 1) <= At(n, 1));
n == At(n, 1);
x == At(x, 1);
y == At(z, 1)
```

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) ==
  At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) ==
  At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
At(n, 1) == At(n, 0);
At(x, 1) == At(x, 0);
At(y, 1) == At(z, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
!(At(z, 1) * At(z, 1) <= At(n, 1));
n == At(n, 1);
x == At(x, 1);
y == At(z, 1)
```

Example: Recursive Square Root Facts

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) == At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) == At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
At(n, 1) == At(n, 0);
At(x, 1) == At(x, 0);
At(y, 1) == At(z, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
!(At(z, 1) * At(z, 1) <= At(n, 1));
At(n, 2) == At(n, 1);
At(x, 2) == At(x, 1);
At(y, 2) == At(z, 1);
!(At(x, 2) + 1 == At(y, 2));
At(z, 2) == (At(x, 2) + At(y, 2)) / 2;
At(z, 2) * At(z, 2) <= At(n, 2);
n == At(n, 2);
x == At(z, 2);
y == At(y, 2)
```

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) == At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) == At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
At(n, 1) == At(n, 0);
At(x, 1) == At(x, 0);
At(y, 1) == At(z, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
!(At(z, 1) * At(z, 1) <= At(n, 1));
At(n, 2) == At(n, 1);
At(x, 2) == At(x, 1);
At(y, 2) == At(z, 1);
!(At(x, 2) + 1 == At(y, 2));
At(z, 2) == (At(x, 2) + At(y, 2)) / 2;
!(At(z, 2) * At(z, 2) <= At(n, 2));
n == At(n, 2);
x == At(x, 2);
y == At(z, 2)
```

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) == At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) == At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
At(n, 1) == At(n, 0);
At(x, 1) == At(x, 0);
At(y, 1) == At(z, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
At(z, 1) * At(z, 1) <= At(n, 1);
At(n, 2) == At(n, 1);
At(x, 2) == At(z, 1);
At(y, 2) == At(y, 1);
!(At(x, 2) + 1 == At(y, 2));
At(z, 2) == (At(x, 2) + At(y, 2)) / 2;
!(At(z, 2) * At(z, 2) <= At(n, 2));
n == At(n, 2);
x == At(x, 2);
y == At(z, 2)
```

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) == At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) == At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
At(n, 1) == At(n, 0);
At(x, 1) == At(x, 0);
At(y, 1) == At(z, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
At(z, 1) * At(z, 1) <= At(n, 1);
At(n, 2) == At(n, 1);
At(x, 2) == At(z, 1);
At(y, 2) == At(y, 1);
!(At(x, 2) + 1 == At(y, 2));
At(z, 2) == (At(x, 2) + At(y, 2)) / 2;
At(z, 2) * At(z, 2) <= At(n, 2);
n == At(n, 2);
x == At(z, 2);
y == At(y, 2)
```

Example: Recursive Square Root Facts

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) == At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) == At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
At(n, 1) == At(n, 0);
At(x, 1) == At(x, 0);
At(y, 1) == At(z, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
!(At(z, 1) * At(z, 1) <= At(n, 1));
At(n, 2) == At(n, 1);
At(x, 2) == At(x, 1);
At(y, 2) == At(z, 1);
!(At(x, 2) + 1 == At(y, 2));
At(z, 2) == (At(x, 2) + At(y, 2)) / 2;
At(z, 2) * At(z, 2) <= At(n, 2);
n == At(n, 2);
x == At(z, 2);
y == At(y, 2)
```

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) == At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) == At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
At(n, 1) == At(n, 0);
At(x, 1) == At(x, 0);
At(y, 1) == At(z, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
!(At(z, 1) * At(z, 1) <= At(n, 1));
At(n, 2) == At(n, 1);
At(x, 2) == At(x, 1);
At(y, 2) == At(z, 1);
!(At(x, 2) + 1 == At(y, 2));
At(z, 2) == (At(x, 2) + At(y, 2)) / 2;
!(At(z, 2) * At(z, 2) <= At(n, 2));
n == At(n, 2);
x == At(x, 2);
y == At(z, 2)
```

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) == At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) == At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
At(n, 1) == At(n, 0);
At(x, 1) == At(x, 0);
At(y, 1) == At(z, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
At(z, 1) * At(z, 1) <= At(n, 1);
At(n, 2) == At(n, 1);
At(x, 2) == At(z, 1);
At(y, 2) == At(y, 1);
!(At(x, 2) + 1 == At(y, 2));
At(z, 2) == (At(x, 2) + At(y, 2)) / 2;
!(At(z, 2) * At(z, 2) <= At(n, 2));
n == At(n, 2);
x == At(x, 2);
y == At(z, 2)
```

```
At[Z]("sq_root_lb.n", 0) >= 0;
At[Z]("sq_root_unfold.n", 0) ==
  At[Z]("sq_root_lb.n", 0);
At(n, 0) == At[Z]("sq_root_unfold.n", 0);
At(x, 0) == 0;
At(y, 0) == At[Z]("sq_root_unfold.n", 0) + 1;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
At(n, 1) == At(n, 0);
At(x, 1) == At(x, 0);
At(y, 1) == At(z, 0);
!(At(x, 1) + 1 == At(y, 1));
At(z, 1) == (At(x, 1) + At(y, 1)) / 2;
At(z, 1) * At(z, 1) <= At(n, 1);
At(n, 2) == At(n, 1);
At(x, 2) == At(z, 1);
At(y, 2) == At(y, 1);
!(At(x, 2) + 1 == At(y, 2));
At(z, 2) == (At(x, 2) + At(y, 2)) / 2;
At(z, 2) * At(z, 2) <= At(n, 2);
n == At(n, 2);
x == At(z, 2);
y == At(y, 2)
```

Termination of the Iterative Square Root

- Termination of the recursion can be expressed by means of a measure

Termination of the Iterative Square Root

- Termination of the recursion can be expressed by means of a measure
- At each recursive call the measure is decreased while the measure is bounded below by 0

Termination of the Iterative Square Root

- Termination of the recursion can be expressed by means of a measure
- At each recursive call the measure is decreased while the measure is bounded below by 0
- As in the iterative implementation,
  we can specify these properties of the measure using assertions

```
@pure def sq_root_rec_unfold_term(n: Z, x: Z, y: Z): Z = {
  val measure_yx_entry: Z = y - x
  assert(measure_yx_entry >= 0)
  if (x + 1 == y) {
    return x
  } else {
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      val measure_yx_call: Z = y - z
      assert(measure_yx_call < measure_yx_entry)
      return sq_root_rec_unfold_term(n, z, y)
    } else {
      val measure_yx_call: Z = z - x
      assert(measure_yx_call < measure_yx_entry)
      return sq_root_rec_unfold_term(n, x, z)
    }
  }
}
```

Termination of the Iterative Square Root

- Termination of the recursion can be expressed by means of a measure
- At each recursive call the measure is decreased while the measure is bounded below by 0
- As in the iterative implementation,
  we can specify these properties of the measure using assertions

```
@pure def sq_root_rec_unfold_term(n: Z, x: Z, y: Z): Z = {
  val measure_yx_entry: Z = y - x
  assert(measure_yx_entry >= 0)
  if (x + 1 == y) {
    return x
  } else {
    val z: Z = (x + y) / 2
    if (z * z <= n) {
      val measure_yx_call: Z = y - z
      assert(measure_yx_call < measure_yx_entry)
      return sq_root_rec_unfold_term(n, z, y)
    } else {
      val measure_yx_call: Z = z - x
      assert(measure_yx_call < measure_yx_entry)
      return sq_root_rec_unfold_term(n, x, z)
    }
  }
}
```

- The corresponding facts including the measures contain the asserted properties

Example: Recursive Square Root Facts

```
At[Z]("sq_root_lb_term.n", 0) >= 0;
At[Z]("sq_root_unfold_term.n", 0) ==
  At[Z]("sq_root_lb_term.n", 0);
At(n, 0) == At[Z]("sq_root_unfold_term.n", 0);
At(x, 0) == 0;
At(y, 0) == At[Z]("sq_root_unfold_term.n", 0) + 1;
At(measure_yx_entry, 0) == At(y, 0) - At(x, 0);
At(measure_yx_entry, 0) >= 0;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
At(z, 0) * At(z, 0) <= At(n, 0);
At(measure_yx_call, 0) == At(y, 0) - At(z, 0);
At(measure_yx_call, 0) < At(measure_yx_entry, 0);
n == At(n, 0);
x == At(z, 0);
y == At(y, 0)
```

```
At[Z]("sq_root_lb_term.n", 0) >= 0;
At[Z]("sq_root_unfold_term.n", 0) ==
  At[Z]("sq_root_lb_term.n", 0);
At(n, 0) == At[Z]("sq_root_unfold_term.n", 0);
At(x, 0) == 0;
At(y, 0) == At[Z]("sq_root_unfold_term.n", 0) + 1;
At(measure_yx_entry, 0) == At(y, 0) - At(x, 0);
At(measure_yx_entry, 0) >= 0;
!(At(x, 0) + 1 == At(y, 0));
At(z, 0) == (At(x, 0) + At(y, 0)) / 2;
!(At(z, 0) * At(z, 0) <= At(n, 0));
At(measure_yx_call, 0) == At(z, 0) - At(x, 0);
At(measure_yx_call, 0) < At(measure_yx_entry, 0);
n == At(n, 0);
x == At(x, 0);
y == At(z, 0)
```

Levels of Assurance

- We have looked at various ways to verify programs

Levels of Assurance

- We have looked at various ways to verify programs
  (1) Full proof: This shows that any execution of a program is correct

Levels of Assurance

- We have looked at various ways to verify programs
  (1) Full proof: This shows that any execution of a program is correct
  (2) Bounded proof: Using unfolding, this shows correctness up to a given depth

Levels of Assurance

- We have looked at various ways to verify programs
  (1) Full proof: This shows that any execution of a program is correct
  (2) Bounded proof: Using unfolding, this shows correctness up to a given depth
  (3) Testing: Using unfolding, this shows correctness for certain values up to a given depth

Levels of Assurance

- We have looked at various ways to verify programs
  (1) Full proof: This shows that any execution of a program is correct
  (2) Bounded proof: Using unfolding, this shows correctness up to a given depth
  (3) Testing: Using unfolding, this shows correctness for certain values up to a given depth
- In practice, one has to judge
  what is the most suitable approach for different parts of software

Levels of Assurance

- We have looked at various ways to verify programs
  (1) Full proof: This shows that any execution of a program is correct
  (2) Bounded proof: Using unfolding, this shows correctness up to a given depth
  (3) Testing: Using unfolding, this shows correctness for certain values up to a given depth
- In practice, one has to judge
  what is the most suitable approach for different parts of software
- In particular, it is a matter of time and effort

Levels of Assurance

- We have looked at various ways to verify programs
  (1) Full proof: This shows that any execution of a program is correct
  (2) Bounded proof: Using unfolding, this shows correctness up to a given depth
  (3) Testing: Using unfolding, this shows correctness for certain values up to a given depth
- In practice, one has to judge
  what is the most suitable approach for different parts of software
- In particular, it is a matter of time and effort
- Using the formal techniques discussed, testing can be made very effective
  by generating test cases from contracts and implementations

Test Cases and Testing

- We have looked at programming at different levels of abstraction

Test Cases and Testing

- We have looked at programming at different levels of abstraction
- High-level programs are often also good specifications

Test Cases and Testing

- We have looked at programming at different levels of abstraction
- High-level programs are often also good specifications
- For these we can generate test cases

Test Cases and Testing

- We have looked at programming at different levels of abstraction
- High-level programs are often also good specifications
- For these we can generate test cases
- Often high-level programs are close to specifications
  and follow the heuristic we have seen in equivalence partitioning

Test Cases and Testing

- We have looked at programming at different levels of abstraction
- High-level programs are often also good specifications
- For these we can generate test cases
- Often high-level programs are close to specifications
  and follow the heuristic we have seen in equivalence partitioning
- So, they will produce good test cases for implementations

Test Cases and Testing

- We have looked at programming at different levels of abstraction
- High-level programs are often also good specifications
- For these we can generate test cases
- Often high-level programs are close to specifications
  and follow the heuristic we have seen in equivalence partitioning
- So, they will produce good test cases for implementations
- Instead of proving an implementation correct with respect to a specification
  we can also generate test cases from the specification
  and use it to test the implementation

Specification, Argumentation, Documentation

- The specification describes the required functionality with precision

Specification, Argumentation, Documentation

- The specification describes the required functionality with precision
- The test cases generated provide evidence to argument for correctness

Specification, Argumentation, Documentation

- The specification describes the required functionality with precision
- The test cases generated provide evidence to argument for correctness
- All of this is contained in the program itself
  to document the correctness argument for others

Specification, Argumentation, Documentation

- The specification describes the required functionality with precision
- The test cases generated provide evidence to argument for correctness
- All of this is contained in the program itself
  to document the correctness argument for others
- In large software projects, the verification methods vary and the argument is complex

Summary

- We have seen how dealing with conditionals (and assignment) is sufficient for testing (and bounded proof)
- We can use termination measures for proof and for testing
- In practice, a mixes of verification techniques are used
- Note, that sometimes testing is necessary,
  e.g., when only some scenarios are known but a complete specification cannot be given