Software Correctness:
The Construction of Correct Software

Loop Unfolding

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

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

From Specification to Implementation

- Let's review the material from last week, linking induction, recursion and iteration

From Specification to Implementation

- Let's review the material from last week, linking induction, recursion and iteration
- We have developed a correct implementation in three steps

From Specification to Implementation

- Let's review the material from last week, linking induction, recursion and iteration
- We have developed a correct implementation in three steps
  (1) Describe the mathematical concepts necessary to express required program properties

From Specification to Implementation

- Let's review the material from last week, linking induction, recursion and iteration

- We have developed a correct implementation in three steps
  (1) Describe the mathematical concepts necessary to express required program properties
  (2) Create a (recursive) specification that is easy to understand (and validate)

From Specification to Implementation

- Let's review the material from last week, linking induction, recursion and iteration

- We have developed a correct implementation in three steps
  (1) Describe the mathematical concepts necessary to express required program properties
  (2) Create a (recursive) specification that is easy to understand (and validate)
  (3) Implement an efficient solution for the specification

From Specification to Implementation

- Let's review the material from last week, linking induction, recursion and iteration
- We have developed a correct implementation in three steps
  (1) Describe the mathematical concepts necessary to express required program properties
  (2) Create a (recursive) specification that is easy to understand (and validate)
  (3) Implement an efficient solution for the specification
- We document clearly **what** is implemented by specifying it in mathematical terms

From Specification to Implementation

- Let's review the material from last week, linking induction, recursion and iteration

- We have developed a correct implementation in three steps
  (1) Describe the mathematical concepts necessary to express required program properties
  (2) Create a (recursive) specification that is easy to understand (and validate)
  (3) Implement an efficient solution for the specification

- We document clearly **what** is implemented by specifying it in mathematical terms
- We document clearly **why** the implementation is correct by supplying a proof

From Specification to Implementation

- Let's review the material from last week, linking induction, recursion and iteration

- We have developed a correct implementation in three steps
  (1) Describe the mathematical concepts necessary to express required program properties
  (2) Create a (recursive) specification that is easy to understand (and validate)
  (3) Implement an efficient solution for the specification

- We document clearly **what** is implemented by specifying it in mathematical terms
- We document clearly **why** the implementation is correct by supplying a proof
- Instead of a proof we also accept other evidence of correctness such as successful tests

From Specification to Implementation

- Let's review the material from last week, linking induction, recursion and iteration

- We have developed a correct implementation in three steps
  (1) Describe the mathematical concepts necessary to express required program properties
  (2) Create a (recursive) specification that is easy to understand (and validate)
  (3) Implement an efficient solution for the specification

- We document clearly **what** is implemented by specifying it in mathematical terms
- We document clearly **why** the implementation is correct by supplying a proof
- Instead of a proof we also accept other evidence of correctness such as successful tests

- Today we make preparations for systematic testing of loops and recursive functions

From Specification to Implementation

- Let's review the material from last week, linking induction, recursion and iteration
- We have developed a correct implementation in three steps
  (1) Describe the mathematical concepts necessary to express required program properties
  (2) Create a (recursive) specification that is easy to understand (and validate)
  (3) Implement an efficient solution for the specification
- We document clearly **what** is implemented by specifying it in mathematical terms
- We document clearly **why** the implementation is correct by supplying a proof
- Instead of a proof we also accept other evidence of correctness such as successful tests
- Today we make preparations for systematic testing of loops and recursive functions
- But first let's refresh our memory of induction, recursion and iteration

Example: Multiplication by Repeated Addition

- We can express multiplication of two natural numbers by repeated addition

Example: Multiplication by Repeated Addition

- We can express multiplication of two natural numbers by repeated addition

$$m * n \;=\; \underbrace{n + n + \ldots + n}_{n \text{ times}}$$

Example: Multiplication by Repeated Addition

- We can express multiplication of two natural numbers by repeated addition
  $m * n \; = \underbrace{n + n + \ldots + n}_{n \text{ times}}$
- The repeated addition on the right-hand side can be expressed recursively

Example: Multiplication by Repeated Addition

- We can express multiplication of two natural numbers by repeated addition
  $m * n \;=\; \underbrace{n + n + \ldots + n}_{n \text{ times}}$
- The repeated addition on the right-hand side can be expressed recursively
  $0 \;*\; n \;=\; 0$

Example: Multiplication by Repeated Addition

- We can express multiplication of two natural numbers by repeated addition
  $$m * n \;=\; \underbrace{n + n + \ldots + n}_{n \text{ times}}$$
- The repeated addition on the right-hand side can be expressed recursively
  $$0 \;*\; n \;=\; 0$$
  $$m' * n \;=\; m * n \;+\; n$$

Example: Multiplication by Repeated Addition

- We can express multiplication of two natural numbers by repeated addition
  $m * n = \underbrace{n + n + \ldots + n}_{n \text{ times}}$
- The repeated addition on the right-hand side can be expressed recursively
  $0 * n = 0$
  $m' * n = m * n + n$
- Using the recursive definition of multiplication we can calculate

Example: Multiplication by Repeated Addition

- We can express multiplication of two natural numbers by repeated addition
  $m * n \;=\; \underbrace{n + n + \ldots + n}_{n \text{ times}}$
- The repeated addition on the right-hand side can be expressed recursively
  $0 \;*\; n \;=\; 0$
  $m' * n \;=\; m * n \;+\; n$
- Using the recursive definition of multiplication we can calculate
  $3 * 1$

Example: Multiplication by Repeated Addition

- We can express multiplication of two natural numbers by repeated addition
  $m * n \; = \; \underbrace{n + n + \ldots + n}_{n \text{ times}}$
- The repeated addition on the right-hand side can be expressed recursively
  $0 \; * \; n \; = \; 0$
  $m' * n \; = \; m * n \; + \; n$
- Using the recursive definition of multiplication we can calculate
  $\quad 3 * 1$
  $= 2 * 1 \; + \; 1$

Example: Multiplication by Repeated Addition

- We can express multiplication of two natural numbers by repeated addition
  $$m * n \; = \; \underbrace{n + n + \ldots + n}_{n \text{ times}}$$
- The repeated addition on the right-hand side can be expressed recursively
  $$\begin{aligned} 0 \; * \; n \; &= \; 0 \\ m' * n \; &= \; m * n \; + \; n \end{aligned}$$
- Using the recursive definition of multiplication we can calculate
  $$\begin{aligned} & \; 3 * 1 \\ = & \; 2 * 1 \; + \; 1 \\ = & \; 1 * 1 \; + \; 1 \; + \; 1 \end{aligned}$$

Example: Multiplication by Repeated Addition

- We can express multiplication of two natural numbers by repeated addition
  $m * n \ = \underbrace{n + n + \ldots + n}_{n \text{ times}}$
- The repeated addition on the right-hand side can be expressed recursively
  $0 \ * \ n \ = \ 0$
  $m' * n \ = \ m * n \ + \ n$
- Using the recursive definition of multiplication we can calculate
  $$\begin{aligned} & 3 * 1 \\ = \ & 2 * 1 \ + \ 1 \\ = \ & 1 * 1 \ + \ 1 \ + \ 1 \\ = \ & 0 * 1 \ + \ 1 \ + \ 1 \ + \ 1 \end{aligned}$$

Example: Multiplication by Repeated Addition

- We can express multiplication of two natural numbers by repeated addition
  $$m * n = \underbrace{n + n + \ldots + n}_{n \text{ times}}$$
- The repeated addition on the right-hand side can be expressed recursively
  $$0 * n = 0$$
  $$m' * n = m * n + n$$
- Using the recursive definition of multiplication we can calculate
  $$\begin{aligned} & 3 * 1 \\ = \; & 2 * 1 + 1 \\ = \; & 1 * 1 + 1 + 1 \\ = \; & 0 * 1 + 1 + 1 + 1 \\ = \; & 0 + 1 + 1 + 1 \end{aligned}$$

Example: Multiplication by Repeated Addition

- We can express multiplication of two natural numbers by repeated addition
  $m * n \;=\; \underbrace{n + n + \ldots + n}_{n \text{ times}}$
- The repeated addition on the right-hand side can be expressed recursively
  $0 \;*\; n \;=\; 0$
  $m' * n \;=\; m * n \;+\; n$
- Using the recursive definition of multiplication we can calculate
  $$\begin{aligned} & 3 * 1 \\ = \; & 2 * 1 \;+\; 1 \\ = \; & 1 * 1 \;+\; 1 \;+\; 1 \\ = \; & 0 * 1 \;+\; 1 \;+\; 1 \;+\; 1 \\ = \; & 0 \;+\; 1 \;+\; 1 \;+\; 1 \end{aligned}$$
- Let's specify it in Slang

Multiplication by Repeated Addition in Slang

- We specify `mult_spec` using the inductive definition of the natural numbers

Multiplication by Repeated Addition in Slang

- We specify `mult_spec` using the inductive definition of the natural numbers

```
@strictpure def mult_spec(m: Z, n: Z): Z = m match {
  case 0 => 0
  case k => mult_spec(k - 1, n) + n
}
```

where `k - 1` denotes the predecessor of natural number `k`

Multiplication by Repeated Addition in Slang

- Induction rules for `mult_spec`:

Multiplication by Repeated Addition in Slang

- Induction rules for `mult_spec`:

  Base case (`m == 0`):

```
@pure def mult_spec_0(n: Z) {
  Contract(
    Ensures(mult_spec(0, n) == 0)
  )
}
```

Multiplication by Repeated Addition in Slang

- Induction rules for `mult_spec`:

Base case (`m == 0`):

```
@pure def mult_spec_0(n: Z) {
  Contract(
    Ensures(mult_spec(0, n) == 0)
  )
}
```

Inductive case (`m > 0`):

```
@pure def mult_spec_step(m: Z, n: Z) {
  Contract(
    Requires(m > 0),
    Ensures(mult_spec(m, n) == mult_spec(m - 1, n) + n)
  )
}
```

Multiplication by Repeated Addition in Slang

- Induction rules for `mult_spec`:

  Base case (`m == 0`):

```
@pure def mult_spec_0(n: Z) {
  Contract(
    Ensures(mult_spec(0, n) == 0)
  )
}
```

  Inductive case (`m > 0`):

```
@pure def mult_spec_step(m: Z, n: Z) {
  Contract(
    Requires(m > 0),
    Ensures(mult_spec(m, n) == mult_spec(m - 1, n) + n)
  )
}
```

- Logika applies these rules automatically

Program Specification

- We can write a specification for a multiplication function:

```
@pure def mult_rec(m: Z, n: Z): Z = {
  Contract(
    Requires(m >= 0),
    Ensures(Res == mult_spec(m, n))
  )
  if (m == 0) {
    return 0
  } else {
    return mult_rec(m - 1, n) + n
  }
}
```

Program Specification

- We can write a specification for a multiplication function:

```
@pure def mult_rec(m: Z, n: Z): Z = {
  Contract(
    Requires(m >= 0),
    Ensures(Res == mult_spec(m, n))
  )
  if (m == 0) {
    return 0
  } else {
    return mult_rec(m - 1, n) + n
  }
}
```

- Of course, in this simple example the structure of the recursive specification resembles closely that of the mathematical definition

Program Specification

- We can write a specification for a multiplication function:

```
@pure def mult_rec(m: Z, n: Z): Z = {
  Contract(
    Requires(m >= 0),
    Ensures(Res == mult_spec(m, n))
  )
  if (m == 0) {
    return 0
  } else {
    return mult_rec(m - 1, n) + n
  }
}
```

- Of course, in this simple example the structure of the recursive specification resembles closely that of the mathematical definition
- As a consequence, Logika proves the post-condition fully-automatically

Program Implementation

- We can implement the program using a while loop

Program Implementation

- We can implement the program using a while loop

```
def mult_it(m: Z, n: Z): Z = {
  Contract(
    Requires(m >= 0),
    Ensures(Res == mult_rec(m, n))
  )
  var i: Z = m
  var k: Z = 0
  while (i > 0) {
    Invariant(
        ...
    )
    ...
  }
  return k
}
```

Program Implementation

- We can implement the program using a while loop

```
def mult_it(m: Z, n: Z): Z = {
  Contract(
    Requires(m >= 0),
    Ensures(Res == mult_rec(m, n))
  )
  var i: Z = m
  var k: Z = 0
  while (i > 0) {
    Invariant(
        ...
    )
    ...
  }
  return k
}
```

where variables `i` and `k` are modified until `k` contains the product

Exercise 1

(A) Implement function `mult_it`
(B) Formulate an invariant
Hint: Use backward conjecture to find a candidate for the invariant
(C) Insert deductions that document why the program is correct
(D) Prove and document that the function terminates

```
def mult_it(m: Z, n: Z): Z = {
  Contract(
    Requires(m >= 0),
    Ensures(Res == mult_rec(m, n))
  )
  var i: Z = m
  var k: Z = 0
  while (i > 0) {
    Invariant(
        ...
    )
    ...
  }
  return k
}
```

Example: Counting Down Recursively

- We can specify counting down recursively as follows

Example: Counting Down Recursively

- We can specify counting down recursively as follows

  $cd(k) =$
  $\quad \textbf{if } k == 0$
  $\quad\quad k$
  $\quad \textbf{else}$
  $\quad\quad cd(k-1)$
- We can calculate $cd(2)$ observing the value of the parameter k at each invocation

Example: Counting Down Recursively

- We can specify counting down recursively as follows

$$
\begin{array}{l}
cd(k) = \\
\quad \textbf{if } k == 0 \\
\qquad k \\
\quad \textbf{else} \\
\qquad cd(k-1)
\end{array}
$$

- We can calculate $cd(2)$ observing the value of the parameter k at each invocation

$$cd(2)$$

Example: Counting Down Recursively

- We can specify counting down recursively as follows

  $cd(k) =$
  $\quad$ **if** $k == 0$
  $\quad\quad k$
  $\quad$ **else**
  $\quad\quad cd(k-1)$

- We can calculate $cd(2)$ observing the value of the parameter k at each invocation

  $\quad cd(2)$
  $\quad\quad \{\ k == 2$ and $k \mathrel{!=} 0\ \}$

Example: Counting Down Recursively

- We can specify counting down recursively as follows

$$
\begin{array}{l}
cd(k) = \\
\quad \textbf{if } k == 0 \\
\quad\quad k \\
\quad \textbf{else} \\
\quad\quad cd(k-1)
\end{array}
$$

- We can calculate $cd(2)$ observing the value of the parameter k at each invocation

$$
\begin{array}{ll}
 & cd(2) \\
 & \quad \{\ k == 2 \text{ and } k \mathrel{!}= 0\ \} \\
= & cd(2-1)
\end{array}
$$

Example: Counting Down Recursively

- We can specify counting down recursively as follows

$$
\begin{array}{l}
cd(k) = \\
\quad \textbf{if } k == 0 \\
\qquad k \\
\quad \textbf{else} \\
\qquad cd(k-1)
\end{array}
$$

- We can calculate $cd(2)$ observing the value of the parameter k at each invocation

$$
\begin{array}{ll}
 & cd(2) \\
 & \quad \{\ k == 2 \text{ and } k \mathrel{!=} 0\ \} \\
= & cd(2-1) \\
 & \quad \{\ k == 2-1 \text{ and } k \mathrel{!=} 0\ \}
\end{array}
$$

Example: Counting Down Recursively

- We can specify counting down recursively as follows

$$
\begin{array}{l}
cd(k) = \\
\quad \textbf{if } k == 0 \\
\qquad k \\
\quad \textbf{else} \\
\qquad cd(k-1)
\end{array}
$$

- We can calculate $cd(2)$ observing the value of the parameter k at each invocation

$$
\begin{array}{ll}
 & cd(2) \\
 & \quad \{\ k == 2 \text{ and } k \mathrel{!}= 0\ \} \\
= & cd(2-1) \\
 & \quad \{\ k == 2-1 \text{ and } k \mathrel{!}= 0\ \} \\
= & cd(2-1-1)
\end{array}
$$

Example: Counting Down Recursively

- We can specify counting down recursively as follows

$$
\begin{array}{l}
cd(k) = \\
\quad \textbf{if } k == 0 \\
\quad\quad k \\
\quad \textbf{else} \\
\quad\quad cd(k-1)
\end{array}
$$

- We can calculate $cd(2)$ observing the value of the parameter k at each invocation

$$
\begin{array}{ll}
 & cd(2) \\
 & \quad \{\ k == 2 \text{ and } k \mathrel{!}= 0\ \} \\
= & cd(2-1) \\
 & \quad \{\ k == 2-1 \text{ and } k \mathrel{!}= 0\ \} \\
= & cd(2-1-1) \\
 & \quad \{\ k == 2-1-1 \text{ and } k == 0\ \}
\end{array}
$$

Example: Counting Down Recursively

- We can specify counting down recursively as follows

$$
\begin{array}{l}
cd(k) = \\
\quad \textbf{if } k == 0 \\
\quad\quad k \\
\quad \textbf{else} \\
\quad\quad cd(k-1)
\end{array}
$$

- We can calculate $cd(2)$ observing the value of the parameter k at each invocation

$$
\begin{array}{ll}
 & cd(2) \\
 & \quad \{\ k == 2 \text{ and } k \mathrel{!=} 0\ \} \\
= & cd(2-1) \\
 & \quad \{\ k == 2-1 \text{ and } k \mathrel{!=} 0\ \} \\
= & cd(2-1-1) \\
 & \quad \{\ k == 2-1-1 \text{ and } k == 0\ \} \\
= & 0
\end{array}
$$

Example: Counting Down Recursively

- We can specify counting down recursively as follows

$$
\begin{array}{l}
cd(k) = \\
\quad \textbf{if } k == 0 \\
\quad\quad k \\
\quad \textbf{else} \\
\quad\quad cd(k-1)
\end{array}
$$

- We can calculate $cd(2)$ observing the value of the parameter k at each invocation

$$
\begin{array}{ll}
 & cd(2) \\
 & \quad \{\ k == 2 \text{ and } k \mathrel{!=} 0\ \} \\
= & cd(2-1) \\
 & \quad \{\ k == 2-1 \text{ and } k \mathrel{!=} 0\ \} \\
= & cd(2-1-1) \\
 & \quad \{\ k == 2-1-1 \text{ and } k == 0\ \} \\
= & 0
\end{array}
$$

Let's rename k at each invocation to clarify what's going on

Example: Counting Down Recursively

- We rename k into $k0$, $k1$, $k2$, ... counting upwards

Example: Counting Down Recursively

- We rename k into $k0$, $k1$, $k2$, ... counting upwards

  $cd(2)$

Example: Counting Down Recursively

- We rename k into $k0$, $k1$, $k2$, ... counting upwards

$$cd(2)$$

$$\{\ k0 == 2 \text{ and } k0\ != 0\ \}$$

Example: Counting Down Recursively

- We rename k into $k0$, $k1$, $k2$, ... counting upwards

$$
\begin{aligned}
& cd(2) \\
& \quad \{\ k0 == 2 \text{ and } k0 \,!\!= 0\ \} \\
= \;& cd(2-1)
\end{aligned}
$$

Example: Counting Down Recursively

- We rename k into $k0$, $k1$, $k2$, ... counting upwards

$$
\begin{aligned}
& cd(2) \\
& \quad \{\ k0 == 2 \text{ and } k0 \mathrel{!=} 0\ \} \\
= \ & cd(2-1) \\
& \quad \{\ k1 == 2-1 \text{ and } k1 \mathrel{!=} 0\ \}
\end{aligned}
$$

Example: Counting Down Recursively

- We rename k into $k0, k1, k2, \ldots$ counting upwards

$$
\begin{aligned}
& cd(2) \\
& \quad \{\ k0 == 2 \text{ and } k0 \mathrel{!=} 0\ \} \\
= \ & cd(2-1) \\
& \quad \{\ k1 == 2-1 \text{ and } k1 \mathrel{!=} 0\ \} \\
= \ & cd(2-1-1)
\end{aligned}
$$

Example: Counting Down Recursively

- We rename k into $k0$, $k1$, $k2$, ... counting upwards

$$
\begin{aligned}
& cd(2) \\
& \quad \{\ k0 == 2 \text{ and } k0\ != 0\ \} \\
= \ & cd(2-1) \\
& \quad \{\ k1 == 2-1 \text{ and } k1\ != 0\ \} \\
= \ & cd(2-1-1) \\
& \quad \{\ k2 == 2-1-1 \text{ and } k2 == 0\ \}
\end{aligned}
$$

Example: Counting Down Recursively

- We rename k into $k0$, $k1$, $k2$, ... counting upwards

$$
\begin{aligned}
& cd(2) \\
& \quad \{\ k0 == 2 \text{ and } k0 \mathrel{!=} 0\ \} \\
= \; & cd(2-1) \\
& \quad \{\ k1 == 2-1 \text{ and } k1 \mathrel{!=} 0\ \} \\
= \; & cd(2-1-1) \\
& \quad \{\ k2 == 2-1-1 \text{ and } k2 == 0\ \} \\
= \; & 0
\end{aligned}
$$

- Now, let's replace sub-expressions by the names of the parameters holding those values

Example: Counting Down Recursively

- Using $k0$, $k1$, $k2$ we get

Example: Counting Down Recursively

- Using $k0$, $k1$, $k2$ we get

  $cd(k0)$

Example: Counting Down Recursively

- Using $k0$, $k1$, $k2$ we get

  $cd(k0)$

  $\{\ k0 == 2$ and $k0\ != 0\ \}$

Example: Counting Down Recursively

- Using $k0$, $k1$, $k2$ we get

$$
\begin{aligned}
& cd(k0) \\
& \quad \{\ k0 == 2 \text{ and } k0 \mathrel{!=} 0\ \} \\
= \ & cd(k0 - 1)
\end{aligned}
$$

Example: Counting Down Recursively

- Using $k0$, $k1$, $k2$ we get

$$
\begin{aligned}
& cd(k0) \\
& \quad \{\ k0 == 2 \text{ and } k0 \mathrel{!=} 0\ \} \\
= \ & cd(k0-1) \\
& \quad \{\ k1 == k0-1 \text{ and } k1 \mathrel{!=} 0\ \}
\end{aligned}
$$

Example: Counting Down Recursively

- Using $k0$, $k1$, $k2$ we get

$$
\begin{aligned}
& cd(k0) \\
& \quad \{\ k0 == 2 \text{ and } k0 \mathrel{!=} 0\ \} \\
= \ & cd(k0-1) \\
& \quad \{\ k1 == k0-1 \text{ and } k1 \mathrel{!=} 0\ \} \\
= \ & cd(k1-1)
\end{aligned}
$$

Example: Counting Down Recursively

- Using $k0$, $k1$, $k2$ we get

$$
\begin{aligned}
& cd(k0) \\
& \quad \{\ k0 == 2 \text{ and } k0\ != 0\ \} \\
= \ & cd(k0 - 1) \\
& \quad \{\ k1 == k0 - 1 \text{ and } k1\ != 0\ \} \\
= \ & cd(k1 - 1) \\
& \quad \{\ k2 == k1 - 1 \text{ and } k2 == 0\ \}
\end{aligned}
$$

Example: Counting Down Recursively

- Using $k0$, $k1$, $k2$ we get

$$
\begin{aligned}
& cd(k0) \\
& \quad \{\ k0 == 2 \text{ and } k0 \mathrel{!=} 0\ \} \\
=\ & cd(k0-1) \\
& \quad \{\ k1 == k0 - 1 \text{ and } k1 \mathrel{!=} 0\ \} \\
=\ & cd(k1-1) \\
& \quad \{\ k2 == k1 - 1 \text{ and } k2 == 0\ \} \\
=\ & k2
\end{aligned}
$$

- Only focusing on the value of the parameter and ignoring the initial value 2, we observe

$$
\begin{aligned}
& k0 \mathrel{!=} 0 \\
& k1 == k0 - 1 \text{ and } k1 \mathrel{!=} 0 \\
& k2 == k1 - 1 \text{ and } k2 == 0
\end{aligned}
$$

Recursively Unfolding Counting Down

- The observation
  $k0 \mathrel{!}= 0$
  $k1 == k0 - 1$ and $k1 \mathrel{!}= 0$
  $k2 == k1 - 1$ and $k2 == 0$
  describes the computation starting with the call $cd(2)$ in terms of the parameter values
- Note, the final $k2 == 0$ which determines that the first branch is chosen and $k2$ is returned
- We can read the function definition as an equation
  $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1),$ (FP1)
- Using lambda notation,
  $cd == \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$ (FP2)
- Theses two equations are called a **fix-point equations**
- Replacing the left-hand side by the right-hand side in either (FP1) or (FP2) is called **unfolding**
- Let's consider (FP2) first and then apply what we've learned to (FP1)

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

  cd

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$\begin{aligned} & cd \\ = \; & \lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } cd(k-1) \end{aligned}$$

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$\begin{aligned} & cd \\ = \; & \lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } cd(k-1) \\ = \; & \lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } cd(k-1))(k-1) \end{aligned}$$

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$
\begin{aligned}
& cd \\
= \; & \lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } cd(k-1) \\
= \; & \lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } cd(k-1))(k-1) \\
= \; & \lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } cd(k-1))(k-1))(k-1)
\end{aligned}
$$

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$
\begin{aligned}
& cd \\
= \; & \lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } cd(k-1) \\
= \; & \lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } cd(k-1))(k-1) \\
= \; & \lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } cd(k-1))(k-1))(k-1)
\end{aligned}
$$

- Let's colour the different k's bound by the lambdas

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$
\begin{aligned}
& cd \\
= \; & \lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } cd(k-1) \\
= \; & \lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } cd(k-1))(k-1) \\
= \; & \lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0) \; k \textbf{ else } cd(k-1))(k-1))(k-1)
\end{aligned}
$$

- Let's colour the different k's bound by the lambdas

$$
\lambda {\color{cyan}k} \cdot \textbf{if } ({\color{cyan}k} == 0) \; {\color{cyan}k} \textbf{ else } (\lambda {\color{magenta}k} \cdot \textbf{if } ({\color{magenta}k} == 0) \; {\color{magenta}k} \textbf{ else } (\lambda {\color{red}k} \cdot \textbf{if } ({\color{red}k} == 0) \; {\color{red}k} \textbf{ else } cd({\color{red}k}-1))({\color{magenta}k}-1))({\color{cyan}k}-1)
$$

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$
\begin{aligned}
& cd \\
= \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1) \\
= \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1) \\
= \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1)
\end{aligned}
$$

- Let's colour the different k's bound by the lambdas

$$
cd2 = \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1)
$$

- Let's call this function $cd2$

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$
\begin{aligned}
& cd \\
= \; & \lambda k \cdot \mathbf{if}\ (k == 0)\ k\ \mathbf{else}\ cd(k-1) \\
= \; & \lambda k \cdot \mathbf{if}\ (k == 0)\ k\ \mathbf{else}\ (\lambda k \cdot \mathbf{if}\ (k == 0)\ k\ \mathbf{else}\ cd(k-1))(k-1) \\
= \; & \lambda k \cdot \mathbf{if}\ (k == 0)\ k\ \mathbf{else}\ (\lambda k \cdot \mathbf{if}\ (k == 0)\ k\ \mathbf{else}\ (\lambda k \cdot \mathbf{if}\ (k == 0)\ k\ \mathbf{else}\ cd(k-1))(k-1))(k-1)
\end{aligned}
$$

- Let's colour the different k's bound by the lambdas

$$cd2 = \lambda k \cdot \mathbf{if}\ (k == 0)\ k\ \mathbf{else}\ (\lambda k \cdot \mathbf{if}\ (k == 0)\ k\ \mathbf{else}\ (\lambda k \cdot \mathbf{if}\ (k == 0)\ k\ \mathbf{else}\ cd(k-1))(k-1))(k-1)$$

- Let's call this function $cd2$
- Now let $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$ be given, and calculate

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$\begin{aligned} & cd \\ = \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1) \\ = \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1) \\ = \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1) \end{aligned}$$

- Let's colour the different k's bound by the lambdas

$$cd2 = \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1)$$

- Let's call this function $cd2$
- Now let $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$ be given, and calculate

$$cd2(k0)$$

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$
\begin{aligned}
& cd \\
= \; & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1) \\
= \; & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1) \\
= \; & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1)
\end{aligned}
$$

- Let's colour the different k's bound by the lambdas

$$cd2 = \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1)$$

- Let's call this function $cd2$
- Now let $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$ be given, and calculate

$$
\begin{aligned}
& cd2(k0) \\
= \; & \textbf{if } (k0 == 0)\ k0 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k0-1)
\end{aligned}
$$

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$\begin{aligned} & cd \\ = \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1) \\ = \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1) \\ = \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1) \end{aligned}$$

- Let's colour the different k's bound by the lambdas

$$cd2 = \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1)$$

- Let's call this function $cd2$
- Now let $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$ be given, and calculate

$$\begin{aligned} & cd2(k0) \\ = \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k0-1) \\ = \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k1) \end{aligned}$$

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$\begin{aligned} & cd \\ = \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1) \\ = \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1) \\ = \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1) \end{aligned}$$

- Let's colour the different k's bound by the lambdas

$$cd2 = \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1)$$

- Let's call this function $cd2$
- Now let $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$ be given, and calculate

$$\begin{aligned} & cd2(k0) \\ = \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k0-1) \\ = \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k1) \\ = \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k1-1) \end{aligned}$$

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$
\begin{aligned}
& cd \\
= \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1) \\
= \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1) \\
= \ & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1)
\end{aligned}
$$

- Let's colour the different k's bound by the lambdas

$$cd2 = \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1)$$

- Let's call this function $cd2$
- Now let $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$ be given, and calculate

$$
\begin{aligned}
& cd2(k0) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k0-1) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k1) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k1-1) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k2)
\end{aligned}
$$

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$
\begin{aligned}
& cd \\
= \; & \lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } cd(k-1) \\
= \; & \lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } cd(k-1))(k-1) \\
= \; & \lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } cd(k-1))(k-1))(k-1)
\end{aligned}
$$

- Let's colour the different k's bound by the lambdas

$$cd2 = \lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } cd(k-1))(k-1))(k-1)$$

- Let's call this function $cd2$
- Now let $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$ be given, and calculate

$$
\begin{aligned}
& cd2(k0) \\
= \; & \textbf{if } (k0 == 0)\; k0 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } cd(k-1))(k-1))(k0-1) \\
= \; & \textbf{if } (k0 == 0)\; k0 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } cd(k-1))(k-1))(k1) \\
= \; & \textbf{if } (k0 == 0)\; k0 \textbf{ else if } (k1 == 0)\; k1 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } cd(k-1))(k1-1) \\
= \; & \textbf{if } (k0 == 0)\; k0 \textbf{ else if } (k1 == 0)\; k1 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\; k \textbf{ else } cd(k-1))(k2) \\
= \; & \textbf{if } (k0 == 0)\; k0 \textbf{ else if } (k1 == 0)\; k1 \textbf{ else if } (k2 == 0)\; k2 \textbf{ else } cd(k2-1)
\end{aligned}
$$

Recursively Unfolding Lambda Using (FP2)

- Unfolding is a calculation that the function itself as a value

$$\begin{aligned} & cd \\ = \; & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1) \\ = \; & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1) \\ = \; & \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1) \end{aligned}$$

- Let's colour the different k's bound by the lambdas

$$cd2 = \lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k-1)$$

- Let's call this function $cd2$
- Now let $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$ be given, and calculate

$$\begin{aligned} & cd2(k0) \\ = \; & \textbf{if } (k0 == 0)\ k0 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k0-1) \\ = \; & \textbf{if } (k0 == 0)\ k0 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k-1))(k1) \\ = \; & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k1-1) \\ = \; & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } (\lambda k \cdot \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1))(k2) \\ = \; & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2-1) \end{aligned}$$

- Let's compare this to our initial observation for the computation of $cd(2)$

Recursive Unfolding Vs Direct Calculation

- Given $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$,

Recursive Unfolding Vs Direct Calculation

- Given $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$, we have

$$\textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2 - 1) \tag{1}$$

Recursive Unfolding Vs Direct Calculation

- Given $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$, we have

$$\textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2 - 1) \tag{1}$$

- The observation
  $k0\ != 0$
  $k1 == k0 - 1$ and $k1\ != 0$
  $k2 == k1 - 1$ and $k2 == 0$

Recursive Unfolding Vs Direct Calculation

- Given $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$, we have

$$\textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2 - 1) \tag{1}$$

- The observation
  $k0\ != 0$
  $k1 == k0 - 1$ and $k1\ != 0$
  $k2 == k1 - 1$ and $k2 == 0$
  describes the situation where expression (1) returns $k2$

Recursive Unfolding Vs Direct Calculation

- Given $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$, we have

$$\textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2 - 1) \tag{1}$$

- The observation
  $k0\ != 0$
  $k1 == k0 - 1$ and $k1\ != 0$
  $k2 == k1 - 1$ and $k2 == 0$
  describes the situation where expression (1) returns $k2$
- This is the case when $k0 == 2$

Recursive Unfolding Vs Direct Calculation

- Given $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$, we have

$$\textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2 - 1) \tag{1}$$

- The observation
  $k0\ != 0$
  $k1 == k0 - 1$ and $k1\ != 0$
  $k2 == k1 - 1$ and $k2 == 0$
  describes the situation where expression (1) returns $k2$
- This is the case when $k0 == 2$
- In other words, when $cd(2)$ is called

Recursive Unfolding Vs Direct Calculation

- Given $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$, we have

$$\textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2 - 1) \tag{1}$$

- The observation
  $k0 \mathrel{!}= 0$
  $k1 == k0 - 1$ and $k1 \mathrel{!}= 0$
  $k2 == k1 - 1$ and $k2 == 0$
  describes the situation where expression (1) returns $k2$
- This is the case when $k0 == 2$
- In other words, when $cd(2)$ is called
- Next let's consider the fix-point equation $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$

Recursive Unfolding Vs Direct Calculation

- Given $k0$, $k1 == k0 - 1$, $k2 == k1 - 1$, we have

$$\textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2 - 1) \tag{1}$$

- The observation
  $k0 \mathrel{!}= 0$
  $k1 == k0 - 1$ and $k1 \mathrel{!}= 0$
  $k2 == k1 - 1$ and $k2 == 0$
  describes the situation where expression (1) returns $k2$
- This is the case when $k0 == 2$
- In other words, when $cd(2)$ is called
- Next let's consider the fix-point equation $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$
- We begin by unfolding it

Unfolding with Parameters using (FP1)

- Using $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$, we calculate

Unfolding with Parameters using (FP1)

- Using $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$, we calculate

$$cd(k0)$$

Unfolding with Parameters using (FP1)

- Using $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$, we calculate

$$
\begin{aligned}
& cd(k0) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k0 - 1)
\end{aligned}
$$

Unfolding with Parameters using (FP1)

- Using $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$, we calculate

$$
\begin{aligned}
& cd(k0) \\
= \; & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k0 - 1)
\end{aligned}
$$

  - Letting $k1 == k0 - 1$

Unfolding with Parameters using (FP1)

- Using $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$, we calculate

$$
\begin{aligned}
& cd(k0) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k0 - 1)
\end{aligned}
$$

  - Letting $k1 == k0 - 1$

$$
= \ \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k1)
$$

Unfolding with Parameters using (FP1)

- Using $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$, we calculate

$$
\begin{aligned}
& cd(k0) \\
= {} & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k0 - 1)
\end{aligned}
$$

  - Letting $k1 == k0 - 1$

$$
\begin{aligned}
= {} & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k1) \\
= {} & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } cd(k1 - 1)
\end{aligned}
$$

Unfolding with Parameters using (FP1)

- Using $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$, we calculate

$$
\begin{aligned}
& cd(k0) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k0 - 1)
\end{aligned}
$$

  - Letting $k1 == k0 - 1$

$$
\begin{aligned}
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k1) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } cd(k1 - 1)
\end{aligned}
$$

  - Letting $k2 == k1 - 1$

Unfolding with Parameters using (FP1)

- Using $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$, we calculate

$$
\begin{aligned}
& cd(k0) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k0 - 1)
\end{aligned}
$$

- Letting $k1 == k0 - 1$

$$
\begin{aligned}
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k1) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } cd(k1 - 1)
\end{aligned}
$$

- Letting $k2 == k1 - 1$

$$
= \ \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } cd(k2)
$$

Unfolding with Parameters using (FP1)

- Using $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$, we calculate

$$
\begin{aligned}
& cd(k0) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k0 - 1)
\end{aligned}
$$

- Letting $k1 == k0 - 1$

$$
\begin{aligned}
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k1) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } cd(k1 - 1)
\end{aligned}
$$

- Letting $k2 == k1 - 1$

$$
\begin{aligned}
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } cd(k2) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2 - 1)
\end{aligned}
$$

Unfolding with Parameters using (FP1)

- Using $cd(k) == \mathbf{if}\ (k == 0)\ k\ \mathbf{else}\ cd(k-1)$, we calculate

$$
\begin{aligned}
& cd(k0) \\
= {} & \mathbf{if}\ (k0 == 0)\ k0\ \mathbf{else}\ cd(k0-1)
\end{aligned}
$$

  - Letting $k1 == k0 - 1$

$$
\begin{aligned}
= {} & \mathbf{if}\ (k0 == 0)\ k0\ \mathbf{else}\ cd(k1) \\
= {} & \mathbf{if}\ (k0 == 0)\ k0\ \mathbf{else}\ \mathbf{if}\ (k1 == 0)\ k1\ \mathbf{else}\ cd(k1-1)
\end{aligned}
$$

  - Letting $k2 == k1 - 1$

$$
\begin{aligned}
= {} & \mathbf{if}\ (k0 == 0)\ k0\ \mathbf{else}\ \mathbf{if}\ (k1 == 0)\ k1\ \mathbf{else}\ cd(k2) \\
= {} & \mathbf{if}\ (k0 == 0)\ k0\ \mathbf{else}\ \mathbf{if}\ (k1 == 0)\ k1\ \mathbf{else}\ \mathbf{if}\ (k2 == 0)\ k2\ \mathbf{else}\ cd(k2-1)
\end{aligned}
$$

- Fix-point equation version (FP2) describes a function as its solution

Unfolding with Parameters using (FP1)

- Using $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$, we calculate

$$
\begin{aligned}
& cd(k0) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k0 - 1)
\end{aligned}
$$

  - Letting $k1 == k0 - 1$

$$
\begin{aligned}
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k1) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } cd(k1 - 1)
\end{aligned}
$$

  - Letting $k2 == k1 - 1$

$$
\begin{aligned}
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } cd(k2) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2 - 1)
\end{aligned}
$$

- Fix-point equation version (FP2) describes a function as its solution
- This function can be used to observe computations via unfolding

Unfolding with Parameters using (FP1)

- Using $cd(k) == \mathbf{if}\ (k == 0)\ k\ \mathbf{else}\ cd(k-1)$, we calculate

$$
\begin{aligned}
& cd(k0) \\
= \ & \mathbf{if}\ (k0 == 0)\ k0\ \mathbf{else}\ cd(k0-1)
\end{aligned}
$$

  - Letting $k1 == k0 - 1$

$$
\begin{aligned}
= \ & \mathbf{if}\ (k0 == 0)\ k0\ \mathbf{else}\ cd(k1) \\
= \ & \mathbf{if}\ (k0 == 0)\ k0\ \mathbf{else}\ \mathbf{if}\ (k1 == 0)\ k1\ \mathbf{else}\ cd(k1-1)
\end{aligned}
$$

  - Letting $k2 == k1 - 1$

$$
\begin{aligned}
= \ & \mathbf{if}\ (k0 == 0)\ k0\ \mathbf{else}\ \mathbf{if}\ (k1 == 0)\ k1\ \mathbf{else}\ cd(k2) \\
= \ & \mathbf{if}\ (k0 == 0)\ k0\ \mathbf{else}\ \mathbf{if}\ (k1 == 0)\ k1\ \mathbf{else}\ \mathbf{if}\ (k2 == 0)\ k2\ \mathbf{else}\ cd(k2-1)
\end{aligned}
$$

- Fix-point equation version (FP2) describes a function as its solution
- This function can be used to observe computations via unfolding
- Fix-point equation version (FP1) can be used directly for unfolding and observation

Unfolding with Parameters using (FP1)

- Using $cd(k) == \textbf{if } (k == 0)\ k \textbf{ else } cd(k-1)$, we calculate

$$
\begin{aligned}
& cd(k0) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k0-1)
\end{aligned}
$$

- Letting $k1 == k0 - 1$

$$
\begin{aligned}
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else } cd(k1) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } cd(k1-1)
\end{aligned}
$$

- Letting $k2 == k1 - 1$

$$
\begin{aligned}
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else } cd(k2) \\
= \ & \textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2-1)
\end{aligned}
$$

- Fix-point equation version (FP2) describes a function as its solution
- This function can be used to observe computations via unfolding
- Fix-point equation version (FP1) can be used directly for unfolding and observation
- It hides the steps involving lambda abstraction and application
- To keep track of consecutive parameter values we introduce new variables at each call

Unfolded Recursive Programs as Facts

- Let's state the expression
  $\textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2 - 1)$
  as a statement where the result value is assigned to a variable Res

Unfolded Recursive Programs as Facts

- Let's state the expression
  $\textbf{if}\ (k0 == 0)\ k0\ \textbf{else if}\ (k1 == 0)\ k1\ \textbf{else if}\ (k2 == 0)\ k2\ \textbf{else}\ cd(k2-1)$
  as a statement where the result value is assigned to a variable Res

$$
\begin{array}{l}
\textbf{if}\ (k0 == 0) \\
\quad Res = k0 \\
\textbf{else} \\
\quad \textbf{if}\ (k1 == 0) \\
\quad\quad Res = k1 \\
\quad \textbf{else} \\
\quad\quad \textbf{if}\ (k2 == 0) \\
\quad\quad\quad Res = k2 \\
\quad\quad \textbf{else} \\
\quad\quad\quad Res = cd(k2-1)
\end{array}
$$

Unfolded Recursive Programs as Facts

- Let's state the expression
  $\textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2-1)$
  as a statement where the result value is assigned to a variable Res

$\textbf{if } (k0 == 0)$	
$\quad Res = k0$	$(k0 == 0 \ => \ Res == k0)\ \&$
$\textbf{else}$	$(k0 \mathrel{!=} 0 \ => \ k1 == k0 - 1)\ \&$
$\quad \textbf{if } (k1 == 0)$	
$\qquad Res = k1$	$(k0 \mathrel{!=} 0\ \&\ k1 == 0 \ => \ Res == k1)\ \&$
$\quad \textbf{else}$	$(k0 \mathrel{!=} 0\ \&\ k1 \mathrel{!=} 0 \ => \ k2 == k1 - 1)\ \&$
$\qquad \textbf{if } (k2 == 0)$	
$\qquad\quad Res = k2$	$(k0 \mathrel{!=} 0\ \&\ k1 \mathrel{!=} 0\ \&\ k2 == 0 \ => \ Res = k2)$
$\qquad \textbf{else}$	$\&$
$\qquad\quad Res = cd(k2-1)$	$(k0 \mathrel{!=} 0\ \&\ k1 \mathrel{!=} 0\ \&\ k2 \mathrel{!=} 0 \ => \ Res == cd(k2-1))$

Unfolded Recursive Programs as Facts

- Let's state the expression
  $\textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2-1)$
  as a statement where the result value is assigned to a variable Res

$\textbf{if } (k0 == 0)$	
$\quad Res = k0$	$\boxed{(k0 == 0 \;\; => \;\; Res == k0)\ \&}$
$\textbf{else}$	$(k0\ != 0 \;\; => \;\; k1 == k0 - 1)\ \&$
$\quad \textbf{if } (k1 == 0)$	
$\quad\quad Res = k1$	$(k0\ != 0\ \&\ k1 == 0 \;\; => \;\; Res == k1)\ \&$
$\quad \textbf{else}$	$\boxed{(k0\ != 0\ \&\ k1\ != 0 \;\; => \;\; k2 == k1 - 1)\ \&}$
$\quad\quad \textbf{if } (k2 == 0)$	
$\quad\quad\quad Res = k2$	$\boxed{(k0\ != 0\ \&\ k1\ != 0\ \&\ k2 == 0 \;\; => \;\; Res = k2)}$
$\quad\quad \textbf{else}$	$\&$
$\quad\quad\quad Res = cd(k2-1)$	$(k0\ != 0\ \&\ k1\ != 0\ \&\ k2\ != 0 \;\; => \;\; Res == cd(k2-1))$

- Within the fact for the unfolded function we also discover our original observation for $cd(2)$

Unfolded Recursive Programs as Facts

- Let's state the expression
  $\textbf{if } (k0 == 0)\ k0 \textbf{ else if } (k1 == 0)\ k1 \textbf{ else if } (k2 == 0)\ k2 \textbf{ else } cd(k2-1)$
  as a statement where the result value is assigned to a variable Res

$\textbf{if } (k0 == 0)$		
$\quad Res = k0$	$\boxed{(k0 == 0 \ \ => \ \ Res == k0)\ \&}$	call $cd(0)$
$\textbf{else}$	$(k0 \mathrel{!=} 0 \ \ => \ \ k1 == k0 - 1)\ \&$	
$\quad \textbf{if } (k1 == 0)$		
$\quad\quad Res = k1$	$(k0 \mathrel{!=} 0\ \&\ k1 == 0 \ \ => \ \ Res == k1)\ \&$	call $cd(1)$
$\quad\quad \textbf{else}$	$\boxed{(k0 \mathrel{!=} 0\ \&\ k1 \mathrel{!=} 0 \ \ => \ \ k2 == k1 - 1)\ \&}$	
$\quad\quad\quad \textbf{if } (k2 == 0)$		
$\quad\quad\quad\quad Res = k2$	$\boxed{(k0 \mathrel{!=} 0\ \&\ k1 \mathrel{!=} 0\ \&\ k2 == 0 \ \ => \ \ Res = k2)}$	
$\quad\quad\quad \textbf{else}$	$\&$	
$\quad\quad\quad\quad Res = cd(k2-1)$	$(k0 \mathrel{!=} 0\ \&\ k1 \mathrel{!=} 0\ \&\ k2 \mathrel{!=} 0 \ \ => \ \ Res == cd(k2-1))$	

- Within the fact for the unfolded function we also discover our original observation for $cd(2)$
- The two shorter cases deal with the calls $cd(0)$ and $cd(1)$

Slang Example: Recursive Counting Down

- The count-down function in Slang:

```
@pure def count0(k: Z): Z = {
  if (k == 0) {
    return k
  } else {
    return count0(k - 1)
  }
}
```

Slang Example: Recursive Counting Down

- The count-down function in Slang:

```
@pure def count0(k: Z): Z = {
  if (k == 0) {
    return k
  } else {
    return count0(k - 1)
  }
}
```

with a separate function specifying its correctness:

```
@pure def count0_0(k: Z): Unit = {
  Contract(
    Requires(k >= 0),
    Ensures(count0(k) == 0)
  )
}
```

- We can unfold function `count0` in Slang

Slang Example: Recursive Counting Down

- The count-down function in Slang:

```
@pure def count0(k: Z): Z = {
  if (k == 0) {
    return k
  } else {
    return count0(k - 1)
  }
}
```

with a separate function specifying its correctness:

```
@pure def count0_0(k: Z): Unit = {
  Contract(
    Requires(k >= 0),
    Ensures(count0(k) == 0)
  )
}
```

- We can unfold function `count0` in Slang
- We do it within the body of the function

Unfolding the Recursive Slang Counting Down Function

- The function itself:

```
@pure def count0(k0: Z): Z = {
  if (k0 == 0) {
    return k0
  } else {
    return count0(k0 - 1)
  }
}
```

Unfolding the Recursive Slang Counting Down Function

- First Unfolding:

```
@pure def count0(k0: Z): Z = {
  if (k0 == 0) {
    return k0
  } else {
    k1 = k0 - 1
    if (k1 == 0) {
      return k1
    } else {
      return count0(k1 - 1)
    }
  }
}
```

Unfolding the Recursive Slang Counting Down Function

- Second Unfolding:

```
@pure def count0(k0: Z): Z = {
  if (k0 == 0) {
    return k0
  } else {
    k1 = k0 - 1
    if (k1 == 0) {
      return k1
    } else {
      k2 = k1 - 1
      if (k2 == 0) {
        return k2
      } else {
        return count0(k2 - 1)
      }
    }
  }
}
```

Unfolding the Recursive Slang Counting Down Function

- Second Unfolding:

```
@pure def count0(k0: Z): Z = {
  if (k0 == 0) {
    return k0
  } else {
    k1 = k0 - 1
    if (k1 == 0) {
      return k1
    } else {
      k2 = k1 - 1
      if (k2 == 0) {
        return k2
      } else {
        return count0(k2 - 1)
      }
    }
  }
}
```

- We can see the effect of recursive unfolding in Slang

Unfolding the Recursive Slang Counting Down Function

- Second Unfolding:

```
@pure def count0(k0: Z): Z = {
  if (k0 == 0) {
    return k0
  } else {
    k1 = k0 - 1
    if (k1 == 0) {
      return k1
    } else {
      k2 = k1 - 1
      if (k2 == 0) {
        return k2
      } else {
        return count0(k2 - 1)
      }
    }
  }
}
```

- We can see the effect of recursive unfolding in Slang
- It occurs when *inter-procedural* check is chosen

Recursive Counting Down and Unfolding in Logika

- Let's inter-procedurally check the post-condition `count0(k) == 0`

```
@pure def count0(k: Z): Z = {
  if (k == 0) {
    return k
  } else {
    return count0(k - 1)
  }
}

@pure def count0_0(k: Z): Unit = {
  Contract(
    Requires(k >= 0),
    Ensures(count0(k) == 0)
  )
}
```

of function `count0_0`

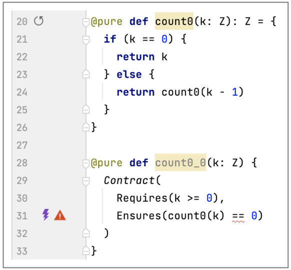
Recursive Counting Down and Unfolding in Logika

- Let's inter-procedurally check the post-condition `count0(k) == 0`

```
@pure def count0(k: Z): Z = {
  if (k == 0) {
    return k
  } else {
    return count0(k - 1)
  }
}

@pure def count0_0(k: Z): Unit = {
  Contract(
    Requires(k >= 0),
    Ensures(count0(k) == 0)
  )
}
```

of function `count0_0`



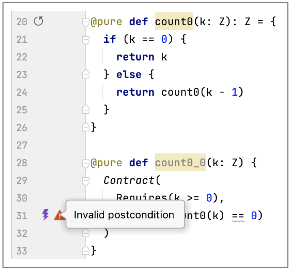
Recursive Counting Down and Unfolding in Logika

- Let's inter-procedurally check the post-condition `count0(k) == 0`

```
@pure def count0(k: Z): Z = {
  if (k == 0) {
    return k
  } else {
    return count0(k - 1)
  }
}

@pure def count0_0(k: Z): Unit = {
  Contract(
    Requires(k >= 0),
    Ensures(count0(k) == 0)
  )
}
```

of function `count0_0`



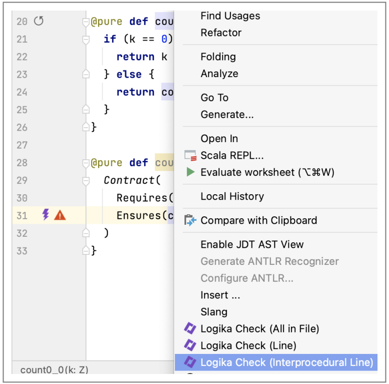
Recursive Counting Down and Unfolding in Logika

- Let's inter-procedurally check the post-condition `count0(k) == 0`

```
@pure def count0(k: Z): Z = {
  if (k == 0) {
    return k
  } else {
    return count0(k - 1)
  }
}

@pure def count0_0(k: Z): Unit = {
  Contract(
    Requires(k >= 0),
    Ensures(count0(k) == 0)
  )
}
```

of function `count0_0`



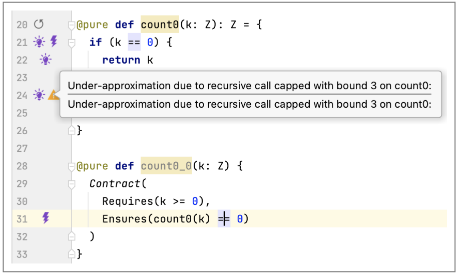
Recursive Counting Down and Unfolding in Logika

- Let's inter-procedurally check the post-condition `count0(k) == 0`

```
@pure def count0(k: Z): Z = {
  if (k == 0) {
    return k
  } else {
    return count0(k - 1)
  }
}

@pure def count0_0(k: Z): Unit = {
  Contract(
    Requires(k >= 0),
    Ensures(count0(k) == 0)
  )
}
```

of function `count0_0`



Recursive Counting Down and Unfolding in Logika

- Let's inter-procedurally check the post-condition `count0(k) == 0`

```
@pure def count0(k: Z): Z = {
  if (k == 0) {
    return k
  } else {
    return count0(k - 1)
  }
}

@pure def count0_0(k: Z): Unit = {
  Contract(
    Requires(k >= 0),
    Ensures(count0(k) == 0)
  )
}
```

of function `count0_0`

```
@pure def count0(k: Z): Z = {
  if (k == 0) {
    return k
```

Under-approximation due to recursive call capped with bound 3 on count0:

Under-approximation due to recursive call capped with bound 3 on count0:

```
}
```

```
{
  At[Z]("count0_0.k", 0) >= 0;
  At(k, 0) == At[Z]("count0_0.k", 0);
  !(At(k, 0) == 0);
  At(k, 1) == At(k, 0) - 1;
  !(At(k, 1) == 0);
  At(k, 2) == At(k, 1) - 1;
  !(At(k, 2) == 0);
  k == At(k, 2) - 1;
  k == 0
}
```

Unfolded if-branch

```
{
  At[Z]("count0_0.k", 0) >= 0;
  At(k, 0) == At[Z]("count0_0.k", 0);
  !(At(k, 0) == 0);
  At(k, 1) == At(k, 0) - 1;
  !(At(k, 1) == 0);
  At(k, 2) == At(k, 1) - 1;
  !(At(k, 2) == 0);
  k == At(k, 2) - 1;
  !(k == 0)
}
```

Unfolded else-branch

Exercise 2: Recursive Factorial Unfolding

(a) Unfold function `fac_rec` two times
(b) Write down the fact for the unfolded function
(c) Inter-procedurally check the post-condition `fac_rec(n) == fac_rec_spec(n)`

```
@pure def fac_rec(n: Z): Z = {
  if (n == 0) {
    return 1
  } else {
    return n * fac_rec(n - 1)
  }
}

@pure def fac_rec_lemma(n: Z) {
  Contract(
    Requires(n >= 0),
    Ensures(fac_rec(n) == fac_rec_spec(n))
  )
}
```

of function `fac_rec_lemma`

Example: Counting Down Iteratively

- We can specify counting down recursively as follows

Example: Counting Down Iteratively

- We can specify counting down recursively as follows

  $cd(k) =$

  $\quad m = k$

  $\quad \textbf{while}\ m > 0$

  $\quad\quad m = m - 1$

  $\quad m$

  where the tailing m is the returned result

Example: Counting Down Iteratively

- We can specify counting down recursively as follows

  $cd(k) =$

  $\quad m = k$

  $\quad \textbf{while } m > 0$

  $\quad\quad m = m - 1$

  $\quad m$

  where the tailing m is the returned result
- We can calculate $cd(2)$ observing the value of the local variable m at each iteration

Example: Counting Down Iteratively

- We can specify counting down recursively as follows

  $cd(k) =$
  $\quad m = k$
  $\quad \textbf{while}\ m > 0$
  $\quad\quad m = m - 1$
  $\quad m$

  where the tailing m is the returned result
- We can calculate $cd(2)$ observing the value of the local variable m at each iteration

  $\quad cd(2)$

Example: Counting Down Iteratively

- We can specify counting down recursively as follows

  $cd(k) =$
  $\quad m = k$
  $\quad \textbf{while } m > 0$
  $\quad\quad m = m - 1$
  $\quad m$

  where the tailing m is the returned result
- We can calculate $cd(2)$ observing the value of the local variable m at each iteration

  $cd(2)$
  $\{\ m == 2 \text{ and } m > 0\ \}$

Example: Counting Down Iteratively

- We can specify counting down recursively as follows

  $cd(k) =$

  $\quad m = k$

  $\quad \textbf{while } m > 0$

  $\quad\quad m = m - 1$

  $\quad m$

  where the tailing m is the returned result
- We can calculate $cd(2)$ observing the value of the local variable m at each iteration

  $cd(2)$

  $\{\ m == 2$ and $m > 0\ \}$

  $\{\ m == 2 - 1$ and $m > 0\ \}$

Example: Counting Down Iteratively

- We can specify counting down recursively as follows

  $cd(k) =$

  $\quad m = k$

  $\quad \textbf{while } m > 0$

  $\quad\quad m = m - 1$

  $\quad m$

  where the tailing m is the returned result
- We can calculate $cd(2)$ observing the value of the local variable m at each iteration

  $cd(2)$

  $\{\ m == 2 \text{ and } m > 0\ \}$

  $\{\ m == 2 - 1 \text{ and } m > 0\ \}$

  $\{\ m == 2 - 1 - 1 \text{ and } m <= 0\ \}$

Example: Counting Down Iteratively

- We can specify counting down recursively as follows

  $cd(k) =$

  $\quad m = k$

  $\quad \textbf{while } m > 0$

  $\qquad m = m - 1$

  $\quad m$

  where the tailing m is the returned result
- We can calculate $cd(2)$ observing the value of the local variable m at each iteration

  $\quad cd(2)$

  $\qquad \{\ m == 2 \text{ and } m > 0\ \}$

  $\qquad \{\ m == 2 - 1 \text{ and } m > 0\ \}$

  $\qquad \{\ m == 2 - 1 - 1 \text{ and } m <= 0\ \}$

  $= \ 0$

Example: Counting Down Iteratively

- We can specify counting down recursively as follows

  $cd(k) =$

  $\quad m = k$

  $\quad \textbf{while}\ m > 0$

  $\quad\quad m = m - 1$

  $\quad m$

  where the tailing m is the returned result
- We can calculate $cd(2)$ observing the value of the local variable m at each iteration

  $\quad cd(2)$

  $\quad\quad \{\ m == 2 \text{ and } m > 0\ \}$

  $\quad\quad \{\ m == 2 - 1 \text{ and } m > 0\ \}$

  $\quad\quad \{\ m == 2 - 1 - 1 \text{ and } m <= 0\ \}$

  $= \ 0$
- It would be convenient

  if we could observe iterative programs similarly to recursive programs

Example: Counting Down Iteratively

- We can specify counting down recursively as follows

  $cd(k) =$
  $\quad m = k$
  $\quad \textbf{while}\ m > 0$
  $\quad\quad m = m - 1$
  $\quad m$

  where the tailing m is the returned result
- We can calculate $cd(2)$ observing the value of the local variable m at each iteration

  $cd(2)$
  $\{\ m == 2 \text{ and } m > 0\ \}$
  $\{\ m == 2 - 1 \text{ and } m > 0\ \}$
  $\{\ m == 2 - 1 - 1 \text{ and } m <= 0\ \}$
  $= \ 0$
- It would be convenient

  if we could observe iterative programs similarly to recursive programs
- Recall the similarity between tail-recursion and while-loops

Example: Counting Down Iteratively

- We rename k into $m0, m1, m2, \ldots$ counting upwards

Example: Counting Down Iteratively

- We rename k into $m0, m1, m2, \ldots$ counting upwards

  $cd(2)$

Example: Counting Down Iteratively

- We rename k into $m0, m1, m2, \ldots$ counting upwards

$$cd(2)$$
$$\{\ m0 == 2 \text{ and } m0 > 0\ \}$$

Example: Counting Down Iteratively

- We rename k into $m0, m1, m2, \ldots$ counting upwards

  $cd(2)$

  $\{ \; m0 == 2$ and $m0 > 0 \; \}$

  $\{ \; m1 == 2 - 1$ and $m1 > 0 \; \}$

Example: Counting Down Iteratively

- We rename k into $m0$, $m1$, $m2$, $\ldots$ counting upwards

  $cd(2)$

  $\{\ m0 == 2$ and $m0 > 0\ \}$

  $\{\ m1 == 2 - 1$ and $m1 > 0\ \}$

  $\{\ m2 == 2 - 1 - 1$ and $m2 <= 0\ \}$

Example: Counting Down Iteratively

- We rename k into $m0$, $m1$, $m2$, ... counting upwards

$$
\begin{aligned}
& cd(2) \\
& \quad \{\ m0 == 2 \text{ and } m0 > 0\ \} \\
& \quad \{\ m1 == 2 - 1 \text{ and } m1 > 0\ \} \\
& \quad \{\ m2 == 2 - 1 - 1 \text{ and } m2 <= 0\ \} \\
= \ & 0
\end{aligned}
$$

Example: Counting Down Iteratively

- We rename k into $m0, m1, m2, \ldots$ counting upwards

$$
\begin{array}{l}
cd(2) \\
\quad \{ \; m0 == 2 \text{ and } m0 > 0 \; \} \\
\quad \{ \; m1 == 2 - 1 \text{ and } m1 > 0 \; \} \\
\quad \{ \; m2 == 2 - 1 - 1 \text{ and } m2 <= 0 \; \} \\
= \; 0
\end{array}
$$

  and replace sub-expressions by variable names

Example: Counting Down Iteratively

- We rename k into $m0$, $m1$, $m2$, ... counting upwards

$$
\begin{array}{l}
\quad cd(2) \\
\qquad \{\ m0 == 2 \text{ and } m0 > 0\ \} \\
\qquad \{\ m1 == 2 - 1 \text{ and } m1 > 0\ \} \\
\qquad \{\ m2 == 2 - 1 - 1 \text{ and } m2 <= 0\ \} \\
= \ 0
\end{array}
$$

  and replace sub-expressions by variable names

$$cd(2)$$

Example: Counting Down Iteratively

- We rename k into $m0$, $m1$, $m2$, ... counting upwards

  $$cd(2)$$
  $$\{\ m0 == 2 \text{ and } m0 > 0\ \}$$
  $$\{\ m1 == 2 - 1 \text{ and } m1 > 0\ \}$$
  $$\{\ m2 == 2 - 1 - 1 \text{ and } m2 <= 0\ \}$$
  $$= \ 0$$

  and replace sub-expressions by variable names

  $$cd(2)$$
  $$\{\ m0 == 2 \text{ and } m0 > 0\ \}$$

Example: Counting Down Iteratively

- We rename k into $m0$, $m1$, $m2$, ... counting upwards

$$
\begin{array}{l}
\quad cd(2) \\
\quad\ \{\ m0 == 2 \text{ and } m0 > 0\ \} \\
\quad\ \{\ m1 == 2 - 1 \text{ and } m1 > 0\ \} \\
\quad\ \{\ m2 == 2 - 1 - 1 \text{ and } m2 <= 0\ \} \\
= \ 0
\end{array}
$$

and replace sub-expressions by variable names

$$
\begin{array}{l}
\quad cd(2) \\
\quad\ \{\ m0 == 2 \text{ and } m0 > 0\ \} \\
\quad\ \{\ m1 == m0 - 1 \text{ and } m1 > 0\ \}
\end{array}
$$

Example: Counting Down Iteratively

- We rename k into $m0$, $m1$, $m2$, $\ldots$ counting upwards

$$
\begin{aligned}
& cd(2) \\
& \quad \{\ m0 == 2 \text{ and } m0 > 0\ \} \\
& \quad \{\ m1 == 2 - 1 \text{ and } m1 > 0\ \} \\
& \quad \{\ m2 == 2 - 1 - 1 \text{ and } m2 <= 0\ \} \\
= \ & 0
\end{aligned}
$$

  and replace sub-expressions by variable names

$$
\begin{aligned}
& cd(2) \\
& \quad \{\ m0 == 2 \text{ and } m0 > 0\ \} \\
& \quad \{\ m1 == m0 - 1 \text{ and } m1 > 0\ \} \\
& \quad \{\ m2 == m1 - 1 \text{ and } m2 <= 0\ \}
\end{aligned}
$$

Example: Counting Down Iteratively

- We rename k into $m0$, $m1$, $m2$, … counting upwards

$$
\begin{array}{l}
\quad cd(2) \\
\quad\ \{\ m0 == 2 \text{ and } m0 > 0\ \} \\
\quad\ \{\ m1 == 2 - 1 \text{ and } m1 > 0\ \} \\
\quad\ \{\ m2 == 2 - 1 - 1 \text{ and } m2 <= 0\ \} \\
= \ 0
\end{array}
$$

  and replace sub-expressions by variable names

$$
\begin{array}{l}
\quad cd(2) \\
\quad\ \{\ m0 == 2 \text{ and } m0 > 0\ \} \\
\quad\ \{\ m1 == m0 - 1 \text{ and } m1 > 0\ \} \\
\quad\ \{\ m2 == m1 - 1 \text{ and } m2 <= 0\ \} \\
= \ 0
\end{array}
$$

- This is exactly the same pattern we have observed for recursion

Example: Counting Down Iteratively

- We rename k into $m0$, $m1$, $m2$, $\ldots$ counting upwards

$$
\begin{array}{l}
\quad cd(2) \\
\quad\ \{\ m0 == 2 \text{ and } m0 > 0\ \} \\
\quad\ \{\ m1 == 2 - 1 \text{ and } m1 > 0\ \} \\
\quad\ \{\ m2 == 2 - 1 - 1 \text{ and } m2 <= 0\ \} \\
= \ 0
\end{array}
$$

  and replace sub-expressions by variable names

$$
\begin{array}{l}
\quad cd(2) \\
\quad\ \{\ m0 == 2 \text{ and } m0 > 0\ \} \\
\quad\ \{\ m1 == m0 - 1 \text{ and } m1 > 0\ \} \\
\quad\ \{\ m2 == m1 - 1 \text{ and } m2 <= 0\ \} \\
= \ 0
\end{array}
$$

- This is exactly the same pattern we have observed for recursion
- Let's look for a fix-point equation

Example: Counting Down Iteratively

- We focus on the iterative part of the body of function cd

$$
\begin{array}{l}
m = k \\
\textbf{while } m > 0 \\
\quad m = m - 1
\end{array}
$$

Example: Counting Down Iteratively

- We focus on the iterative part of the body of function cd

  $m = k$

  $\textbf{while}\ m > 0$

  $\quad m = m - 1$

- To observe one step of the execution of the loop we consider the following

Example: Counting Down Iteratively

- We focus on the iterative part of the body of function cd

  $m = k$

  $\textbf{while } m > 0$

  $\quad m = m - 1$

- To observe one step of the execution of the loop we consider the following
  - If the condition $m > 0$ is true, we execute the loop body and the execute the loop again

    $m = m - 1; \textbf{ while } (m > 0) \; m = m - 1$

Example: Counting Down Iteratively

- We focus on the iterative part of the body of function cd

  $m = k$
  while $m > 0$
  $\quad m = m - 1$
- To observe one step of the execution of the loop we consider the following
  - If the condition $m > 0$ is true, we execute the loop body and the execute the loop again
    $m = m - 1$; **while** $(m > 0)$ $m = m - 1$
  - If the condition is false, the loop is exited
    (and the statement following the loop may be executed)

Example: Counting Down Iteratively

- We focus on the iterative part of the body of function cd

  $m = k$

  $\textbf{while } m > 0$

  $\quad m = m - 1$
- To observe one step of the execution of the loop we consider the following
  - If the condition $m > 0$ is true, we execute the loop body and the execute the loop again

    $m = m - 1; \textbf{ while } (m > 0) \; m = m - 1$
  - If the condition is false, the loop is exited

    (and the statement following the loop may be executed)
- The above describes a conditional with an empty else-branch

Example: Counting Down Iteratively

- We focus on the iterative part of the body of function cd

  $m = k$

  $\mathbf{while}\ m > 0$

  $\quad m = m - 1$

- To observe one step of the execution of the loop we consider the following
  - If the condition $m > 0$ is true, we execute the loop body and the execute the loop again

    $m = m - 1;\ \mathbf{while}\ (m > 0)\ m = m - 1$
  - If the condition is false, the loop is exited

    (and the statement following the loop may be executed)
- The above describes a conditional with an empty else-branch
- We have

  $$\mathbf{while}\ (m > 0)\ m = m - 1\ \ ==\ \ \mathbf{if}\ (m > 0)\ \{\ m = m - 1;\ \mathbf{while}\ (m > 0)\ m = m - 1\ \} \qquad \text{(FP3)}$$

Example: Counting Down Iteratively

- We focus on the iterative part of the body of function cd

  $m = k$
  $\textbf{while}\ m > 0$
  $\quad m = m - 1$
- To observe one step of the execution of the loop we consider the following
  - If the condition $m > 0$ is true, we execute the loop body and the execute the loop again
    $m = m - 1;\ \textbf{while}\ (m > 0)\ m = m - 1$
  - If the condition is false, the loop is exited
    (and the statement following the loop may be executed)
- The above describes a conditional with an empty else-branch
- We have (in colour)

  $\textbf{while}\ (m > 0)\ m = m - 1 \ == \ \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ \textbf{while}\ (m > 0)\ m = m - 1\ \}$ (FP3)

Example: Counting Down Iteratively

- We focus on the iterative part of the body of function cd

  $m = k$
  $\textbf{while}\ m > 0$
  $\quad m = m - 1$
- To observe one step of the execution of the loop we consider the following
  - If the condition $m > 0$ is true, we execute the loop body and the execute the loop again
    $m = m - 1;\ \textbf{while}\ (m > 0)\ m = m - 1$
  - If the condition is false, the loop is exited
    (and the statement following the loop may be executed)
- The above describes a conditional with an empty else-branch
- We have (in colour)

$$\textbf{while}\ (m > 0)\ m = m - 1\ \ == \ \ \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ \textbf{while}\ (m > 0)\ m = m - 1\ \} \qquad \text{(FP3)}$$
- The loop is a solution of fix-point equation (FP3)

Example: Counting Down Iteratively

- We focus on the iterative part of the body of function cd

  $m = k$

  $\textbf{while}\ m > 0$

  $\quad m = m - 1$
- To observe one step of the execution of the loop we consider the following
  - If the condition $m > 0$ is true, we execute the loop body and the execute the loop again

    $m = m - 1;\ \textbf{while}\ (m > 0)\ m = m - 1$
  - If the condition is false, the loop is exited

    (and the statement following the loop may be executed)
- The above describes a conditional with an empty else-branch
- We have (in colour)

  $\textbf{while}\ (m > 0)\ m = m - 1 \ == \ \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ \textbf{while}\ (m > 0)\ m = m - 1\ \}$ (FP3)
- The loop is a solution of fix-point equation (FP3)
- We can use it for unfolding while-loops

Loop Unfolding using (FP3)

- Using
  $\mathbf{while}\ (m > 0)\ m = m - 1\ \ ==\ \ \mathbf{if}\ (m > 0)\ \{\ m = m - 1;\ \mathbf{while}\ (m > 0)\ m = m - 1\ \}$,
  abbreviating $\mathbf{while}\ (m > 0)\ m = m - 1$ with W, we calculate

Loop Unfolding using (FP3)

- Using
  $\mathbf{while}\ (m > 0)\ m = m - 1\ \ ==\ \ \mathbf{if}\ (m > 0)\ \{\, m = m - 1;\ \mathbf{while}\ (m > 0)\ m = m - 1 \,\}$,
  abbreviating $\mathbf{while}\ (m > 0)\ m = m - 1$ with W, we calculate

  $\mathbf{while}\ (m > 0)\ m = m - 1$

Loop Unfolding using (FP3)

- Using
  $\mathbf{while}\ (m > 0)\ m = m - 1 \ == \ \mathbf{if}\ (m > 0)\ \{\ m = m - 1;\ \mathbf{while}\ (m > 0)\ m = m - 1\ \}$,
  abbreviating $\mathbf{while}\ (m > 0)\ m = m - 1$ with W, we calculate

$$
\begin{aligned}
& \mathbf{while}\ (m > 0)\ m = m - 1 \\
= \ & \mathbf{if}\ (m > 0)\ \{\ m = m - 1;\ W\ \}
\end{aligned}
$$

Loop Unfolding using (FP3)

- Using
  $\mathbf{while}\ (m > 0)\ m = m - 1\ \ ==\ \ \mathbf{if}\ (m > 0)\ \{\, m = m - 1;\ \mathbf{while}\ (m > 0)\ m = m - 1 \,\}$,
  abbreviating $\mathbf{while}\ (m > 0)\ m = m - 1$ with W, we calculate

$$
\begin{aligned}
& \mathbf{while}\ (m > 0)\ m = m - 1 \\
= \ & \mathbf{if}\ (m > 0)\ \{\, m = m - 1;\ W \,\} \\
= \ & \mathbf{if}\ (m > 0)\ \{\, m = m - 1;\ \mathbf{if}\ (m > 0)\ \{\, m = m - 1;\ W \,\}\,\}
\end{aligned}
$$

Loop Unfolding using (FP3)

- Using
  $\textbf{while}\ (m > 0)\ m = m - 1\ \ {==}\ \ \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ \textbf{while}\ (m > 0)\ m = m - 1\ \}$,
  abbreviating $\textbf{while}\ (m > 0)\ m = m - 1$ with W, we calculate

$$
\begin{aligned}
& \textbf{while}\ (m > 0)\ m = m - 1 \\
= {} & \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ W\ \} \\
= {} & \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ W\ \}\ \} \\
= {} & \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ W\ \}\ \}\ \}
\end{aligned}
$$

Loop Unfolding using (FP3)

- Using
  $\mathbf{while}\ (m > 0)\ m = m - 1\ \ ==\ \ \mathbf{if}\ (m > 0)\ \{\ m = m - 1;\ \mathbf{while}\ (m > 0)\ m = m - 1\ \}$,
  abbreviating $\mathbf{while}\ (m > 0)\ m = m - 1$ with W, we calculate

$$
\begin{aligned}
& \mathbf{while}\ (m > 0)\ m = m - 1 \\
=\ & \mathbf{if}\ (m > 0)\ \{\ m = m - 1;\ W\ \} \\
=\ & \mathbf{if}\ (m > 0)\ \{\ m = m - 1;\ \mathbf{if}\ (m > 0)\ \{\ m = m - 1;\ W\ \}\ \} \\
=\ & \mathbf{if}\ (m > 0)\ \{\ m = m - 1;\ \mathbf{if}\ (m > 0)\ \{\ m = m - 1;\ \mathbf{if}\ (m > 0)\ \{\ m = m - 1;\ W\ \}\ \}\ \}
\end{aligned}
$$

- More readable this is

$$
\begin{array}{l}
\mathbf{if}\ (m > 0) \\
\quad m = m - 1 \\
\quad \mathbf{if}\ (m > 0) \\
\quad\quad m = m - 1 \\
\quad\quad \mathbf{if}\ (m > 0) \\
\quad\quad\quad m = m - 1 \\
\quad\quad\quad W
\end{array}
$$

Loop Unfolding using (FP3)

- Using
  $\textbf{while}\ (m > 0)\ m = m - 1\ \ ==\ \ \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ \textbf{while}\ (m > 0)\ m = m - 1\ \}$,
  abbreviating $\textbf{while}\ (m > 0)\ m = m - 1$ with W, we calculate

$$
\begin{aligned}
& \textbf{while}\ (m > 0)\ m = m - 1 \\
= \ & \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ W\ \} \\
= \ & \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ W\ \}\ \} \\
= \ & \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ \textbf{if}\ (m > 0)\ \{\ m = m - 1;\ W\ \}\ \}\ \}
\end{aligned}
$$

- More readable this is

$$
\begin{array}{l}
\textbf{if}\ (m > 0) \\
\quad m = m - 1 \\
\quad \textbf{if}\ (m > 0) \\
\quad\quad m = m - 1 \\
\quad\quad \textbf{if}\ (m > 0) \\
\quad\quad\quad m = m - 1 \\
\quad\quad\quad W
\end{array}
$$

...and as a fact

$$
\begin{array}{l}
(m0 <= 0\ \ => \ \ m == m0)\ \& \\
(m0 > 0\ \ => \ \ m1 == m0 - 1)\ \& \\
(m0 > 0\ \&\ m1 <= 0\ \ => \ \ m == m1)\ \& \\
(m0 > 0\ \&\ m1 > 0\ \ => \ \ m2 = m1 - 1)\ \& \\
(m0 > 0\ \&\ m1 > 0\ \&\ m1 <= 0\ \ => \ \ m == m2)\ \& \\
\ldots
\end{array}
$$

Loop Unfolding using (FP3)

- The complete body of the loop unfolded twice:
  $m0 == k$
  $(m0 <= 0 \;\; => \;\; m == m0)$ &
  $(m0 > 0 \;\; => \;\; m1 == m0 - 1)$ &
  $(m0 > 0 \;\&\; m1 <= 0 \;\; => \;\; m == m1)$ &
  $(m0 > 0 \;\&\; m1 > 0 \;\; => \;\; m2 = m1 - 1)$ &
  $(m0 > 0 \;\&\; m1 > 0 \;\&\; m1 <= 0 \;\; => \;\; m == m2)$ &
  $Res == m$

- Note the similarity of the structure of the formula with respect to the variables $m0$, $m1$, $m2$ in the iterative case and the variables $k0$, $k1$, $k2$ in the recursive case
  - In the iterative case the variables occur as a consequence of consecutive assignments
  - In the recursive case they occur as a consequence of consecutive parameter passing

Unfolding the Iterative Slang Counting Down Function

- The function itself:

```
@pure def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m
}
```

Unfolding the Iterative Slang Counting Down Function

- First Unfolding:

```
@pure def while0(k: Z): Z = {
  var m: Z = k
  if (m > 0) {
    m = m - 1
    while (m > 0) {
      m = m - 1
    }
  }
  return m
}
```

Unfolding the Iterative Slang Counting Down Function

- Second Unfolding:

```
@pure def while0(k: Z): Z = {
  var m: Z = k
  if (m > 0) {
    m = m - 1
    if (m > 0) {
      m = m - 1
      while (m > 0) {
        m = m - 1
      }
    }
  }
  return m
}
```

Unfolding the Iterative Slang Counting Down Function

- Second Unfolding:

```
@pure def while0(k: Z): Z = {
  var m: Z = k
  if (m > 0) {
    m = m - 1
    if (m > 0) {
      m = m - 1
      while (m > 0) {
        m = m - 1
      }
    }
  }
  return m
}
```

- We can see the effect of iterative unfolding in Slang

Unfolding the Iterative Slang Counting Down Function

- Second Unfolding:

```
@pure def while0(k: Z): Z = {
  var m: Z = k
  if (m > 0) {
    m = m - 1
    if (m > 0) {
      m = m - 1
      while (m > 0) {
        m = m - 1
      }
    }
  }
  return m
}
```

- We can see the effect of iterative unfolding in Slang
- It occurs when *inter-procedural* check is chosen

Iterative Counting Down and Unfolding in Logika

- Let's inter-procedurally check the post-condition `while0(k) == 0`

```
@pure def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m
}

@pure def while0_0(k: Z) {
  Contract(
    Requires(k >= 0),
    Ensures(while0(k) == 0)
  )
}
```

of function `while0_0`

Iterative Counting Down and Unfolding in Logika

- Let's inter-procedurally check the post-condition `while0(k) == 0`

```
@pure def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m
}

@pure def while0_0(k: Z) {
  Contract(
    Requires(k >= 0),
    Ensures(while0(k) == 0)
  )
}
```

```
@pure def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m
}

@pure def while0_0(k: Z) {
  Contract(
    Requires(k >= 0),
    Ensures(while0(k) == 0)
  )
}
```

of function `while0_0`

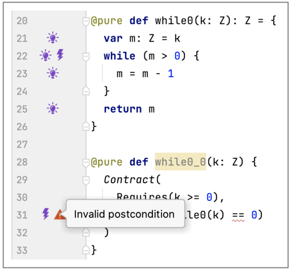
Iterative Counting Down and Unfolding in Logika

- Let's inter-procedurally check the post-condition `while0(k) == 0`

```
@pure def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m
}

@pure def while0_0(k: Z) {
  Contract(
    Requires(k >= 0),
    Ensures(while0(k) == 0)
  )
}
```

of function `while0_0`



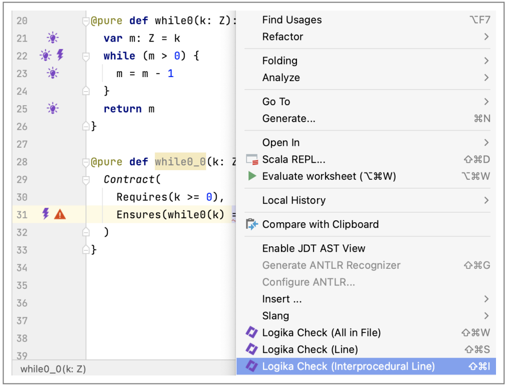
Iterative Counting Down and Unfolding in Logika

- Let's inter-procedurally check the post-condition `while0(k) == 0`

```
@pure def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m
}

@pure def while0_0(k: Z) {
  Contract(
    Requires(k >= 0),
    Ensures(while0(k) == 0)
  )
}
```

of function `while0_0`



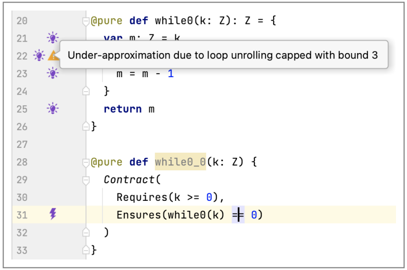
Iterative Counting Down and Unfolding in Logika

- Let's inter-procedurally check the post-condition `while0(k) == 0`

```
@pure def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m
}

@pure def while0_0(k: Z) {
  Contract(
    Requires(k >= 0),
    Ensures(while0(k) == 0)
  )
}
```

of function `while0_0`



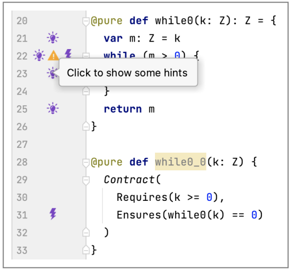
Iterative Counting Down and Unfolding in Logika

- Let's inter-procedurally check the post-condition `while0(k) == 0`

```
@pure def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m
}

@pure def while0_0(k: Z) {
  Contract(
    Requires(k >= 0),
    Ensures(while0(k) == 0)
  )
}
```

of function `while0_0`



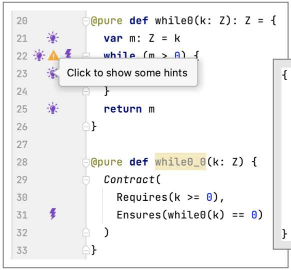
Iterative Counting Down and Unfolding in Logika

- Let's inter-procedurally check the post-condition `while0(k) == 0`

```
@pure def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m
}

@pure def while0_0(k: Z) {
  Contract(
    Requires(k >= 0),
    Ensures(while0(k) == 0)
  )
}
```

of function `while0_0`



Second Unfolding

```
{
  At[Z]("while0_0.k", 0) >= 0;
  k == At[Z]("while0_0.k", 0);
  At(m, 0) == k;
  At(m, 0) > 0;
  At(m, 1) == At(m, 0) - 1;
  At(m, 1) > 0;
  At(m, 2) == At(m, 1) - 1;
  At(m, 2) > 0;
  m == At(m, 2) - 1;
  m > 0
}
```

Exercise 3: Iterative Factorial Unfolding

(a) Unfold the loop of the function `fac_it` two times
(b) Write down the fact for the unfolded function
(c) Inter-procedurally check the post-condition `fac_it(n) == fac_rec(n)`

```
@pure def fac_it(n: Z): Z = {
  var x: Z = 1
  var m: Z = 0;
  while (m < n) {
    m = m + 1
    x = x * m
  }
  return x
}

@pure def fac_it_rec_lemma(n: Z) {
  Contract(
    Requires(n >= 0),
    Ensures(fac_it(n) == fac_rec(n))
  )
}
```

of function `fac_it_rec_lemma`

Symbolic Execution with Recursion

```
def count0(k: Z): Z = {
  if (k == 0) {
    return k // Res = k
  } else {
    return count0(k - 1)
  }
}
```

Symbolic Execution with Recursion

```
def count0(k: Z): Z = {
  if (k == 0) {
    return k // Res = k
  } else {
    return count0(k - 1)
  }
}
```

- Executing `count0(k)` yields (`k: K0`), (PC: **true**)

Symbolic Execution with Recursion

```
def count0(k: Z): Z = {
  if (k == 0) {
    return k // Res = k
  } else {
    return count0(k - 1)
  }
}
```

- Executing `count0(k)` yields (`k: K0`), (PC: **`true`**)
- Executing `if (k == 0) {` yields (`k: K0`), (PC: `K0 == 0`)

Symbolic Execution with Recursion

```
def count0(k: Z): Z = {
  if (k == 0) {
    return k // Res = k
  } else {
    return count0(k - 1)
  }
}
```

- Executing `count0(k)` yields (`k: K0`), (PC: **`true`**)
- Executing `if (k == 0) {` yields (`k: K0`), (PC: `K0 == 0`)
- Executing `return k` yields (`k: K0`, `Res: K0`), (PC: `K0 == 0`)

Symbolic Execution with Recursion

```
def count0(k: Z): Z = {
  if (k == 0) {
    return k // Res = k
  } else {
    return count0(k - 1)
  }
}
```

- Executing `count0(k)` yields (`k: K0`), (PC: `true`)
- Executing `if (k == 0) {` yields (`k: K0`), (PC: `K0 == 0`)
- Executing `return k` yields (`k: K0`, `Res: K0`), (PC: `K0 == 0`)

- Executing `count0(k)` yields (`k: K0`), (PC: `true`)

Symbolic Execution with Recursion

```
def count0(k: Z): Z = {
  if (k == 0) {
    return k // Res = k
  } else {
    return count0(k - 1)
  }
}
```

- Executing `count0(k)` yields (`k: K0`), (PC: `true`)
- Executing `if (k == 0) {` yields (`k: K0`), (PC: `K0 == 0`)
- Executing `return k` yields (`k: K0`, `Res: K0`), (PC: `K0 == 0`)

- Executing `count0(k)` yields (`k: K0`), (PC: `true`)
- Executing `} else {` yields (`k: K0`), (PC: `K0 != 0`)

Symbolic Execution with Recursion

```
def count0(k: Z): Z = {
  if (k == 0) {
    return k // Res = k
  } else {
    return count0(k - 1)
  }
}
```

- Executing `count0(k)` yields (`k: K0`), (PC: `true`)
- Executing `if (k == 0) {` yields (`k: K0`), (PC: `K0 == 0`)
- Executing `return k` yields (`k: K0`, `Res: K0`), (PC: `K0 == 0`)

- Executing `count0(k)` yields (`k: K0`), (PC: `true`)
- Executing `} else {` yields (`k: K0`), (PC: `K0 != 0`)
- Executing `return count0(k - 1)` yields (`k: K0`), (PC: `K0 != 0`, `K1 == K0 - 1`)

Symbolic Execution with Recursion

```
def count0(k: Z): Z = {
  if (k == 0) {
    return k // Res = k
  } else {
    return count0(k - 1)
  }
}
```

- Executing `count0(k)` yields (`k: K0`), (PC: `true`)
- Executing `if (k == 0) {` yields (`k: K0`), (PC: `K0 == 0`)
- Executing `return k` yields (`k: K0`, `Res: K0`), (PC: `K0 == 0`)

- Executing `count0(k)` yields (`k: K0`), (PC: `true`)
- Executing `} else {` yields (`k: K0`), (PC: `K0 != 0`)
- Executing `return count0(k - 1)` yields (`k: K0`), (PC: `K0 != 0`, `K1 == K0 - 1`)
- Executing `count0(k)` yields (`k: K1`), (PC: `K0 != 0`, `K1 == K0 - 1`)

Symbolic Execution with Recursion

```
def count0(k: Z): Z = {
  if (k == 0) {
    return k // Res = k
  } else {
    return count0(k - 1)
  }
}
```

- Executing `count0(k)` yields (`k: K0`), (PC: `true`)
- Executing `if (k == 0) {` yields (`k: K0`), (PC: `K0 == 0`)
- Executing `return k` yields (`k: K0`, `Res: K0`), (PC: `K0 == 0`)

- Executing `count0(k)` yields (`k: K0`), (PC: `true`)
- Executing `} else {` yields (`k: K0`), (PC: `K0 != 0`)
- Executing `return count0(k - 1)` yields (`k: K0`), (PC: `K0 != 0`, `K1 == K0 - 1`)
- Executing `count0(k)` yields (`k: K1`), (PC: `K0 != 0`, `K1 == K0 - 1`)
- Executing `if (k == 0) {` yields (`k: K1`), (PC: `K0 != 0`, `K1 == K0 - 1`, `K1 == 0`)

Symbolic Execution with Recursion

```
def count0(k: Z): Z = {
  if (k == 0) {
    return k // Res = k
  } else {
    return count0(k - 1)
  }
}
```

- Executing `count0(k)` yields (`k: K0`), (PC: `true`)
- Executing `if (k == 0) {` yields (`k: K0`), (PC: `K0 == 0`)
- Executing `return k` yields (`k: K0`, `Res: K0`), (PC: `K0 == 0`)

- Executing `count0(k)` yields (`k: K0`), (PC: `true`)
- Executing `} else {` yields (`k: K0`), (PC: `K0 != 0`)
- Executing `return count0(k - 1)` yields (`k: K0`), (PC: `K0 != 0`, `K1 == K0 - 1`)
- Executing `count0(k)` yields (`k: K1`), (PC: `K0 != 0`, `K1 == K0 - 1`)
- Executing `if (k == 0) {` yields (`k: K1`), (PC: `K0 != 0`, `K1 == K0 - 1`, `K1 == 0`)
- Executing `return k` yields (`k: K1`, `Res: K1`), (PC: `K0 != 0`, `K1 == K0 - 1`, `K1 == 0`)

Symbolic Execution with Iteration

```
def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m // Res = m
}
```

Symbolic Execution with Iteration

```
def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m // Res = m
}
```

- Executing `while0(k)` yields (`k: K`), (PC: **true**)

Symbolic Execution with Iteration

```
def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m // Res = m
}
```

- Executing `while0(k)` yields (`k: K`), (PC: **true**)
- Executing `var m: Z = k` yields (`k: K`, `m: K`), (PC: **true**)

Symbolic Execution with Iteration

```
def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m // Res = m
}
```

- Executing `while0(k)` yields (`k: K`), (PC: **`true`**)
- Executing **`var`** `m: Z = k` yields (`k: K`, `m: K`), (PC: **`true`**)
- Executing `}` yields (`k: K`, `m: K`), (PC: `K <= 0`)

Symbolic Execution with Iteration

```
def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m // Res = m
}
```

- Executing `while0(k)` yields (`k: K`), (PC: **`true`**)
- Executing `var m: Z = k` yields (`k: K`, `m: K`), (PC: **`true`**)
- Executing `}` yields (`k: K`, `m: K`), (PC: `K <= 0`)
- Executing `return m` yields (`k: K`, `m: K`, `Res: K`), (PC: `K <= 0`)

Symbolic Execution with Iteration

```
def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m // Res = m
}
```

- Executing `while0(k)` yields (`k: K`), (PC: `true`)
- Executing `var m: Z = k` yields (`k: K`, `m: K`), (PC: `true`)
- Executing `}` yields (`k: K`, `m: K`), (PC: `K <= 0`)
- Executing `return m` yields (`k: K`, `m: K`, `Res: K`), (PC: `K <= 0`)

- Executing `while0(k)` yields (`k: K`), (PC: `true`)

Symbolic Execution with Iteration

```
def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m // Res = m
}
```

- Executing `while0(k)` yields (`k: K`), (PC: `true`)
- Executing `var m: Z = k` yields (`k: K`, `m: K`), (PC: `true`)
- Executing `}` yields (`k: K`, `m: K`), (PC: `K <= 0`)
- Executing `return m` yields (`k: K`, `m: K`, `Res: K`), (PC: `K <= 0`)

- Executing `while0(k)` yields (`k: K`), (PC: `true`)
- Executing `var m: Z = k` yields (`k: K`, `m: K`), (PC: `true`)

Symbolic Execution with Iteration

```
def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m // Res = m
}
```

- Executing `while0(k)` yields (`k: K`), (PC: **`true`**)
- Executing `var m: Z = k` yields (`k: K`, `m: K`), (PC: **`true`**)
- Executing `}` yields (`k: K`, `m: K`), (PC: `K <= 0`)
- Executing `return m` yields (`k: K`, `m: K`, `Res: K`), (PC: `K <= 0`)

- Executing `while0(k)` yields (`k: K`), (PC: **`true`**)
- Executing `var m: Z = k` yields (`k: K`, `m: K`), (PC: **`true`**)
- Executing `while (m > 0) {` yields (`k: K`, `m: K`), (PC: `K > 0`)

Symbolic Execution with Iteration

```
def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m // Res = m
}
```

- Executing `while0(k)` yields (`k: K`), (PC: `true`)
- Executing `var m: Z = k` yields (`k: K`, `m: K`), (PC: `true`)
- Executing `}` yields (`k: K`, `m: K`), (PC: `K <= 0`)
- Executing `return m` yields (`k: K`, `m: K`, `Res: K`), (PC: `K <= 0`)

- Executing `while0(k)` yields (`k: K`), (PC: `true`)
- Executing `var m: Z = k` yields (`k: K`, `m: K`), (PC: `true`)
- Executing `while (m > 0) {` yields (`k: K`, `m: K`), (PC: `K > 0`)
- Executing `m = m - 1` yields (`k: K`, `m: M1`), (PC: `K > 0`, `M1 > K - 1`)

Symbolic Execution with Iteration

```
def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m // Res = m
}
```

- Executing `while0(k)` yields (`k: K`), (PC: `true`)
- Executing `var m: Z = k` yields (`k: K`, `m: K`), (PC: `true`)
- Executing `}` yields (`k: K`, `m: K`), (PC: `K <= 0`)
- Executing `return m` yields (`k: K`, `m: K`, `Res: K`), (PC: `K <= 0`)

- Executing `while0(k)` yields (`k: K`), (PC: `true`)
- Executing `var m: Z = k` yields (`k: K`, `m: K`), (PC: `true`)
- Executing `while (m > 0) {` yields (`k: K`, `m: K`), (PC: `K > 0`)
- Executing `m = m - 1` yields (`k: K`, `m: M1`), (PC: `K > 0`, `M1 > K - 1`)
- Executing `}` yields (`k: K`, `m: M1`), (PC: `K > 0`, `M1 > K - 1`, `M1 <= 0`)

Symbolic Execution with Iteration

```
def while0(k: Z): Z = {
  var m: Z = k
  while (m > 0) {
    m = m - 1
  }
  return m // Res = m
}
```

- Executing `while0(k)` yields (`k: K`), (PC: `true`)
- Executing `var m: Z = k` yields (`k: K`, `m: K`), (PC: `true`)
- Executing `}` yields (`k: K`, `m: K`), (PC: `K <= 0`)
- Executing `return m` yields (`k: K`, `m: K`, `Res: K`), (PC: `K <= 0`)

- Executing `while0(k)` yields (`k: K`), (PC: `true`)
- Executing `var m: Z = k` yields (`k: K`, `m: K`), (PC: `true`)
- Executing `while (m > 0) {` yields (`k: K`, `m: K`), (PC: `K > 0`)
- Executing `m = m - 1` yields (`k: K`, `m: M1`), (PC: `K > 0`, `M1 > K - 1`)
- Executing `}` yields (`k: K`, `m: M1`), (PC: `K > 0`, `M1 > K - 1`, `M1 <= 0`)
- Executing `return m` yields (`k: K`, `m: M1`, `Res: M1`), (PC: `K > 0`, `M1 > K - 1`, `M1 <= 0`)

Summary

- We have reviewed development and verification methodology for Slang programs
- We have looked at unfolding of recursive functions
- We have looked at unfolding of while-loops
- We have considered fix-points that provide a justification for unfolding
- We have looked at symbolic execution of recursive functions
- We have looked at symbolic execution of while-loops