Verification of Imperative Programs in Scala

Régis Blanc

Supervisor: Prof. Viktor Kuncak
Referee: Prof. Andrey Rybalchenko
TA: Philippe Suter

Laboratory for Automated Reasoning and Analysis
École Polytechnique Fédérale de Lausanne

July 19, 2012

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Is This Correct?

```
def binarySearch(a: Array[Int], key: Int): Int = {

  var low = 0
  var high = a.length - 1
  var res = -1
  (while(low < high && res == -1) {
    val i = (high + low) / 2
    val v = a(i)
    if(v == key) res = i
    else if(v > key) high = i - 1
    else if(v < key) low = i + 1
  })




  res
}
```

Is This Correct?

```
def binarySearch(a: Array[Int], key: Int): Int = {
  require(a.length > 0 && sorted(a, 0, a.length - 1))
  var low = 0
  var high = a.length − 1
  var res = −1
  (while(low < high && res == −1) {
    val i = (high + low) / 2
    val v = a(i)
    if(v == key) res = i
    else if(v > key) high = i − 1
    else if(v < key) low = i + 1
  })




  res
} ensuring(res => res >= -1 && res < a.length && (
                   if(res >= 0) a(res) == key else !occurs(a, 0, a.length, key)))
```

Is This Correct?

```
def binarySearch(a: Array[Int], key: Int): Int = {
  require(a.length > 0 && sorted(a, 0, a.length - 1))
  var low = 0
  var high = a.length − 1
  var res = −1
  (while(low < high && res == −1) {
    val i = (high + low) / 2
    val v = a(i)
    if (v == key) res = i
    else if (v > key) high = i − 1
    else if (v < key) low = i + 1
  })




  res
} ensuring(res => res >= -1 && res < a.length && (
                  if(res >= 0) a(res) == key else !occurs(a, 0, a.length, key)))
```

Counterexample: `a = Array(971), key = 971`

Is This Correct?

```
def binarySearch(a: Array[Int], key: Int): Int = {
  require(a.length > 0 && sorted(a, 0, a.length - 1))
  var low = 0
  var high = a.length − 1
  var res = −1
  (while(low <= high && res == −1) {
    val i = (high + low) / 2
    val v = a(i)
    if(v == key) res = i
    else if(v > key) high = i − 1
    else if(v < key) low = i + 1
  })




  res
} ensuring(res => res >= -1 && res < a.length && (
                  if(res >= 0) a(res) == key else !occurs(a, 0, a.length, key)))
```

Is This Correct?

```
def binarySearch(a: Array[Int], key: Int): Int = {

  var low = 0
  var high = a.length − 1
  var res = −1
  (while(low <= high && res == −1) {
    val i = (high + low) / 2
    val v = a(i)
    if (v == key) res = i
    else if (v > key) high = i − 1
    else if (v < key) low = i + 1
  })




  res
} ensuring(res => res >= -1 && res < a.length && (
                  if(res >= 0) a(res) == key else !occurs(a, 0, a.length, key)))
```

Counterexample: `a = Array(609, 608), key = 608`

Is This Correct?

```
def binarySearch(a: Array[Int], key: Int): Int = {
  require(a.length > 0 && sorted(a, 0, a.length - 1))
  var low = 0
  var high = a.length − 1
  var res = −1
  (while(low <= high && res == −1) {
    val i = (high + low) / 2
    val v = a(i)
    if (v == key) res = i
    else if (v > key) high = i − 1
    else if (v < key) low = i + 1
  }) invariant(res >= −1 && res < a.length && 0 <= low &&
               low <= high && high >= 1 && high < a.length &&
               if (res >= 0) a(res) == key
               else (!occurs(a, 0, low, key) &&
                     !occurs(a, high + 1, a.length, key)))
  res
} ensuring(res => res >= -1 && res < a.length && (
                 if(res >= 0) a(res) == key else !occurs(a, 0, a.length, key)))
```

Is This Correct?

```
def binarySearch(a: Array[Int], key: Int): Int = {
  require(a.length > 0 && sorted(a, 0, a.length - 1))
  var low = 0
  var high = a.length − 1
  var res = −1
  (while(low <= high && res == −1) {
    val i = (high + low) / 2
    val v = a(i)
    if(v == key) res = i
    else if(v > key) high = i − 1
    else if(v < key) low = i + 1
  }) invariant(res >= −1 && res < a.length && 0 <= low &&
                low <= high && high >= 1 && high < a.length &&
                if (res >= 0) a(res) == key
                else (!occurs(a, 0, low, key) &&
                      !occurs(a, high + 1, a.length, key)))
  res
} ensuring(res => res >= -1 && res < a.length && (
                  if(res >= 0) a(res) == key else !occurs(a, 0, a.length, key)))
```

Counterexample:

`a = Array(2571, 2571), high = 1, low = 1, key = 2572`

Is This Correct?

```
def binarySearch(a: Array[Int], key: Int): Int = {
  require(a.length > 0 && sorted(a, 0, a.length - 1))
  var low = 0
  var high = a.length − 1
  var res = −1
  (while(low <= high && res == −1) {
    val i = (high + low) / 2
    val v = a(i)
    if(v == key) res = i
    else if(v > key) high = i − 1
    else if(v < key) low = i + 1
  }) invariant(res >= −1 && res < a.length && 0 <= low &&
               low <= high + 1 && high >= 1 && high < a.length &&
               if (res >= 0) a(res) == key
               else (!occurs(a, 0, low, key) &&
                     !occurs(a, high + 1, a.length, key)))
  res
} ensuring(res => res >= -1 && res < a.length && (
                 if(res >= 0) a(res) == key else !occurs(a, 0, a.length, key)))
```

The Leon Verification System

- Verifier for the Scala language.
- Original developer: Philippe Suter.
- Supported a well-defined functional subset of Scala.
  - Now also handling imperative code.

```
def fact(n: Int): Int = {
  if(n == 0)
    1
  else
    n * fact(n-1)
}
```

```
def size(t: Tree): Int = t match {
  case Node(left, _, right) =>
    1 + size(left) + size(right)
  case Leaf() => 0
}
```

- Complete for finding counterexamples.

Contracts

Specifications can be defined using contracts.

Contracts

Specifications can be defined using contracts.

- Postconditions

```
def abs(n: Int): Int = {
  if(n <= 0) -n else n
} ensuring(res => res >= 0)
```

Contracts

Specifications can be defined using contracts.

- Postconditions

```
def abs(n: Int): Int = {
  if(n <= 0) -n else n
} ensuring(res => res >= 0)
```

- Preconditions

```
def fact(n: Int): Int = {
  require(n >= 0)
  if(n == 0) 1 else n * fact(n-1)
}
```

Contracts

Specifications can be defined using contracts.

- Postconditions

```
def abs(n: Int): Int = {
  if(n <= 0) -n else n
} ensuring(res => res >= 0)
```

- Preconditions

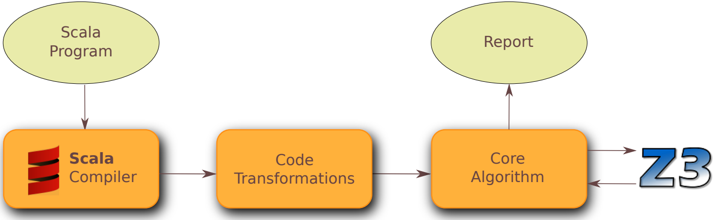
```
def fact(n: Int): Int = {
  require(n >= 0)
  if(n == 0) 1 else n * fact(n-1)
}
```

The implementation and specification languages are the same.

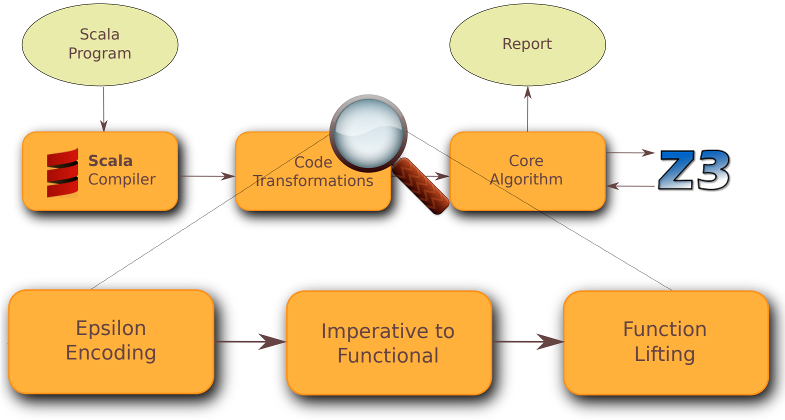
Contributions of This Thesis

- Extend Leon to enable verification of imperative programs.
  - While loops,
  - Local variables and assignments, and
  - Sequential composition (blocks).
- Framework to automatically generate testcases in Leon.
- Support for non-deterministic programs.
- Enrichment of the input language:
  - Additional data types (`Tuples`, `Array`, `List`), and
  - Additional native operations (`modulo`, `instanceof`).

Our Architecture: Verification of Imperative Programs



Our Architecture: Verification of Imperative Programs



Static Single Assignment Form


```
a = 1
b = 0
a = a + 1
b = a + b
a = a + b
```

Static Single Assignment Form


```
a = 1
b = 0
a = a + 1
b = a + b
a = a + b
```

```
val a = 1
val b = 0
```

Static Single Assignment Form


```
a = 1
b = 0
a = a + 1
b = a + b
a = a + b
```

⟹

```
val a = 1
val b = 0
val a1 = a + 1
```

Static Single Assignment Form


```
a = 1
b = 0
a = a + 1
b = a + b
a = a + b
```

→

```
val a = 1
val b = 0
val a1 = a + 1
val b1 = a1 + b
```

Static Single Assignment Form


```
a = 1
b = 0
a = a + 1
b = a + b
a = a + b
```

```scala
val a = 1
val b = 0
val a1 = a + 1
val b1 = a1 + b
val a2 = a1 + b1
```

If Expressions


```
if(x < 0)
  a = a + 1
else
  b = b + 3
```

If Expressions


```
if(x < 0)
  a = a + 1
else
  b = b + 3
```

```
val (a1, b1) =
```

If Expressions


```
if(x < 0)
  a = a + 1
else
  b = b + 3
```

```
val (a1, b1) = if(x < 0) {

} else {

}
```

If Expressions


```
if(x < 0)
  a = a + 1
else
  b = b + 3
```

```
val (a1, b1) = if(x < 0) {
  (a + 1, b)
} else {

}
```

If Expressions


```
if(x < 0)
  a = a + 1
else
  b = b + 3
```

```
val (a1, b1) = if(x < 0) {
  (a + 1, b)
} else {
  (a, b + 3)
}
```

Loops as Recusive Functions


```
(while(i < n) {
  i = i + 1
  s = s + i
})
```

Loops as Recusive Functions


```
(while(i < n) {
  i = i + 1
  s = s + i
})
```

```
def rec(i: Int, s: Int):
    (Int, Int) = {




}
```

Loops as Recusive Functions


```
(while(i < n) {
  i = i + 1
  s = s + i
})
```

```
def rec(i: Int, s: Int):
    (Int, Int) = {

  if(i < n)

  else

}
```

Loops as Recusive Functions


```
(while(i < n) {
  i = i + 1
  s = s + i
})
```

```
def rec(i: Int, s: Int):
    (Int, Int) = {

  if(i < n)
    rec(i+1, s+i+1)
  else

}
```

Loops as Recusive Functions


```
(while(i < n) {
  i = i + 1
  s = s + i
})
```

```
def rec(i: Int, s: Int):
    (Int, Int) = {

  if(i < n)
    rec(i+1, s+i+1)
  else
    (i, s)
}
```

Loops as Recusive Functions


```
(while(i < n) {
  i = i + 1
  s = s + i
}) invariant(s >= 0)
```

```
def rec(i: Int, s: Int):
    (Int, Int) = {

  if(i < n)
    rec(i+1, s+i+1)
  else
    (i, s)
}
```

Loops as Recusive Functions


```
(while(i < n) {
  i = i + 1
  s = s + i
}) invariant(s >= 0)
```

```
def rec(i: Int, s: Int):
    (Int, Int) = {
  require(s >= 0)
  if(i < n)
    rec(i+1, s+i+1)
  else
    (i, s)
}
```

Loops as Recusive Functions


```
(while(i < n) {
  i = i + 1
  s = s + i
}) invariant(s >= 0)
```

```
def rec(i: Int, s: Int):
    (Int, Int) = {
  require(s >= 0)
  if(i < n)
    rec(i+1, s+i+1)
  else
    (i, s)
} ensuring(_._2 >= 0)
```

Lifting of Nested Functions


```
def foo(a: Int) = {
  require(a > 0)

  def rec(b: Int) =
    a + b

  rec(a + 1)
}
```

Lifting of Nested Functions


```scala
def foo(a: Int) = {
  require(a > 0)

  def rec(b: Int) =
    a + b

  rec(a + 1)
}
```

```scala
def foo(a: Int) = {
  require(a > 0)

  def rec(a1:  Int, b:  Int) = {

    a1 + b
  }
  rec(a, a + 1)
}




}
```

Lifting of Nested Functions


```
def foo(a: Int) = {
  require(a > 0)

  def rec(b: Int) =
    a + b

  rec(a + 1)
}
```

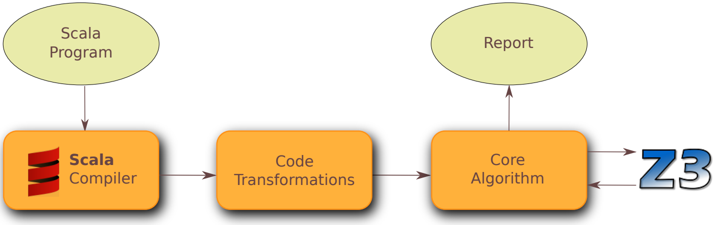
```
def foo(a: Int) = {
  require(a > 0)

  def rec(a1: Int, b: Int) = {
    require(a1 > 0)
    a1 + b
  }
  rec(a, a + 1)
}



}
```

Lifting of Nested Functions

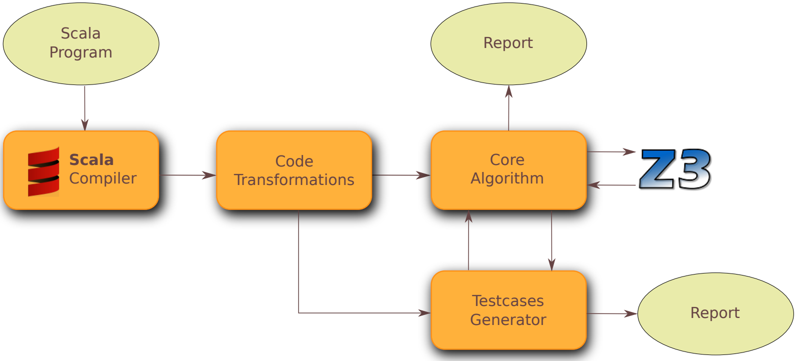

```
def foo(a: Int) = {
  require(a > 0)

  def rec(b: Int) =
    a + b

  rec(a + 1)
}
```

```
def foo(a: Int) = {
  require(a > 0)




  }
  rec(a, a + 1)
}
def rec(a1: Int, b: Int) = {
    require(a1 > 0)
    a1 + b
}
```

Our Architecture: Automatic Generation of Testcases

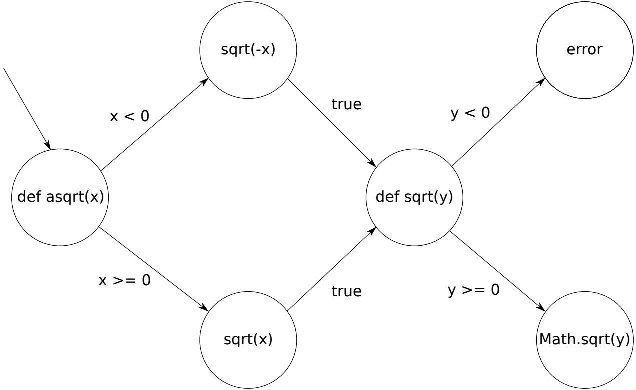


Our Architecture: Automatic Generation of Testcases



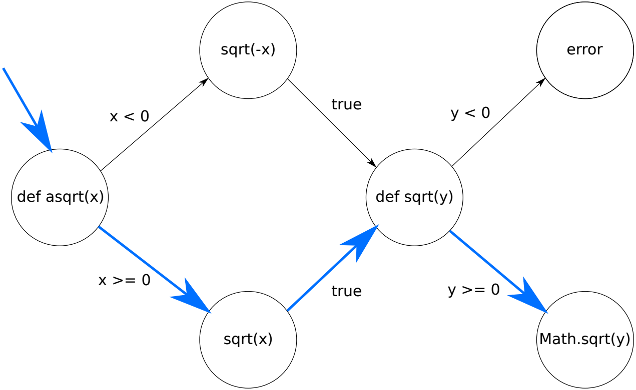
Automatic Generation of Testcases for Coverage

```
def asqrt(x: Int) = if(x < 0) sqrt(-x) else sqrt(x)
def sqrt(y: Int) = if(y < 0) error else Math.sqrt(y)
```



Automatic Generation of Testcases for Coverage

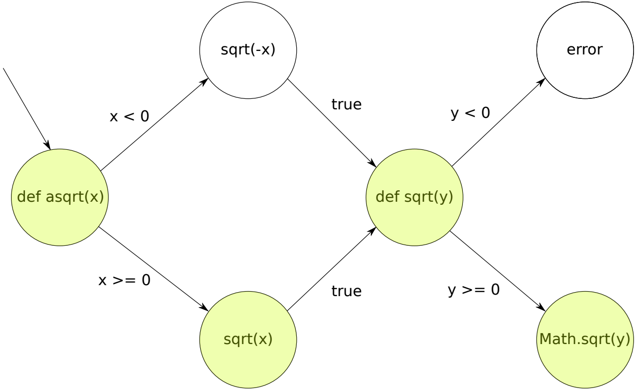
```
def asqrt(x: Int) = if(x < 0) sqrt(-x) else sqrt(x)
def sqrt(y: Int) = if(y < 0) error else Math.sqrt(y)
```



$x \geq 0 \wedge x \geq 0$ Testcase: $x = 0$

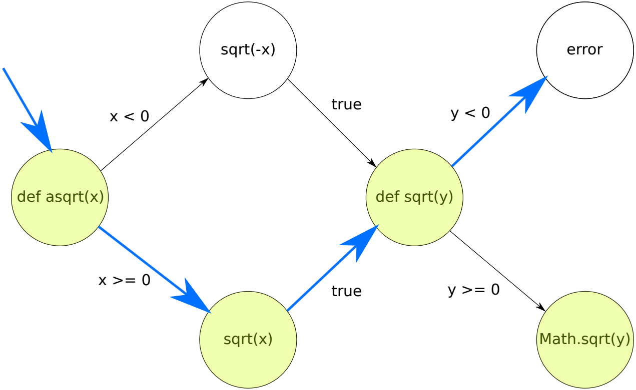
Automatic Generation of Testcases for Coverage

```
def asqrt(x: Int) = if(x < 0) sqrt(-x) else sqrt(x)
def sqrt(y: Int) = if(y < 0) error else Math.sqrt(y)
```



Automatic Generation of Testcases for Coverage

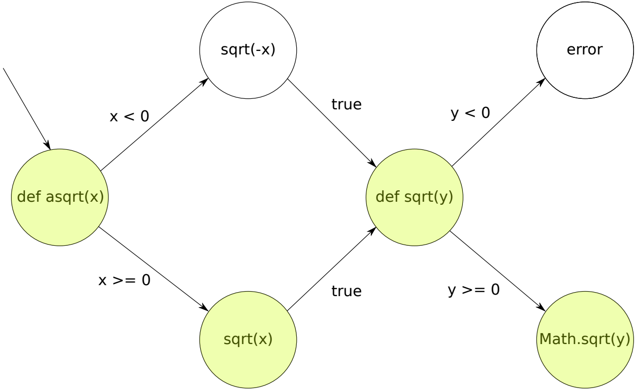
```
def asqrt(x: Int) = if(x < 0) sqrt(-x) else sqrt(x)
def sqrt(y: Int) = if(y < 0) error else Math.sqrt(y)
```



$x \geq 0 \land x < 0$ Unsatisfiable

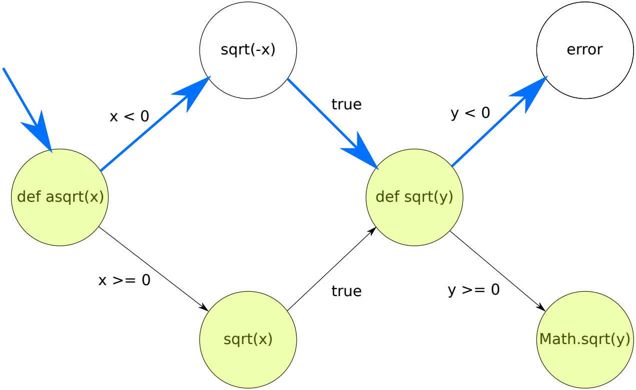
Automatic Generation of Testcases for Coverage

```
def asqrt(x: Int) = if(x < 0) sqrt(-x) else sqrt(x)
def sqrt(y: Int) = if(y < 0) error else Math.sqrt(y)
```



Automatic Generation of Testcases for Coverage

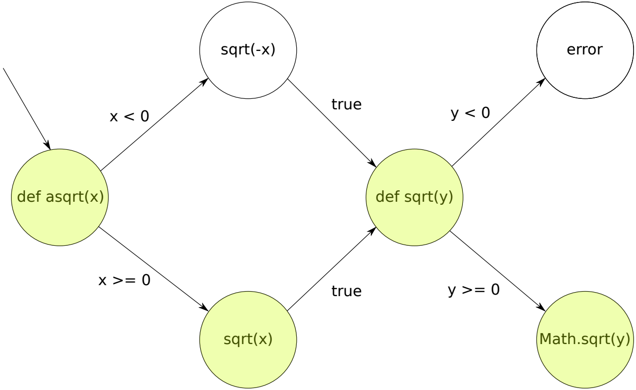
```
def asqrt(x: Int) = if(x < 0) sqrt(-x) else sqrt(x)
def sqrt(y: Int) = if(y < 0) error else Math.sqrt(y)
```



$x < 0 \wedge -x < 0$ Unsatisfiable

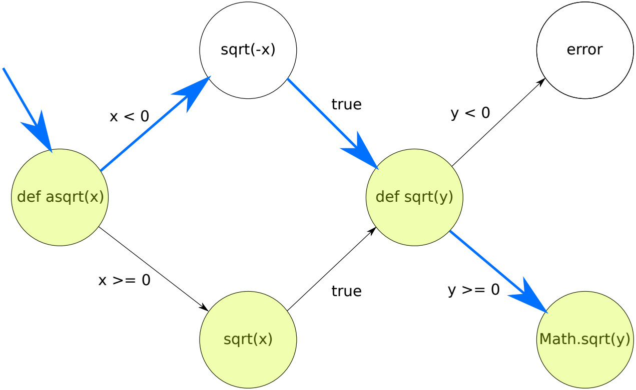
Automatic Generation of Testcases for Coverage

```
def asqrt(x: Int) = if(x < 0) sqrt(-x) else sqrt(x)
def sqrt(y: Int) = if(y < 0) error else Math.sqrt(y)
```



Automatic Generation of Testcases for Coverage

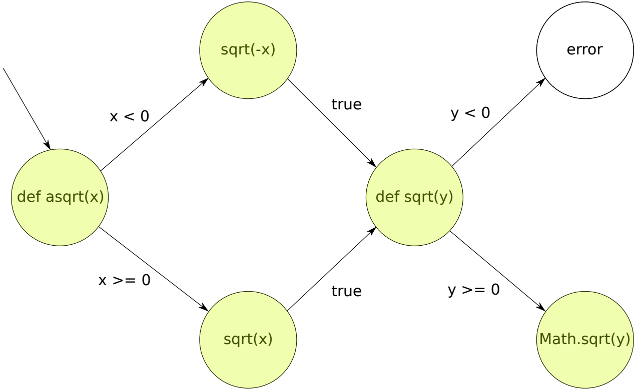
```
def asqrt(x: Int) = if(x < 0) sqrt(-x) else sqrt(x)
def sqrt(y: Int) = if(y < 0) error else Math.sqrt(y)
```



$x < 0 \wedge -x \geq 0$ Testcase: $x = -1$

Automatic Generation of Testcases for Coverage

```
def asqrt(x: Int) = if(x < 0) sqrt(-x) else sqrt(x)
def sqrt(y: Int) = if(y < 0) error else Math.sqrt(y)
```



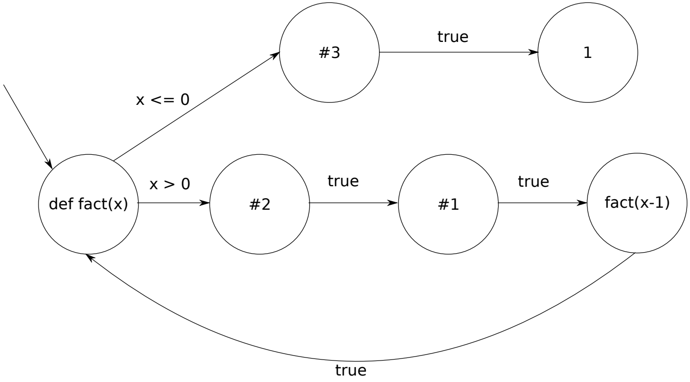
Testcases: $x = 0, x = -1$

`error` unreachable

Generation of Deeper Testcases using Waypoints

Finding a path that visits each waypoint in order.

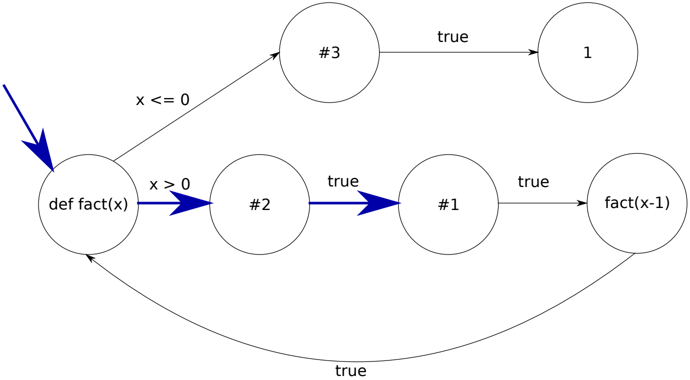
```
def fact(x: Int) = if(x <= 0) 1#3 else x*f(x-1)#1,#2
```



Generation of Deeper Testcases using Waypoints

Finding a path that visits each waypoint in order.

```
def fact(x: Int) = if(x <= 0) 1^{#3} else x*f(x-1)^{#1,#2}
```

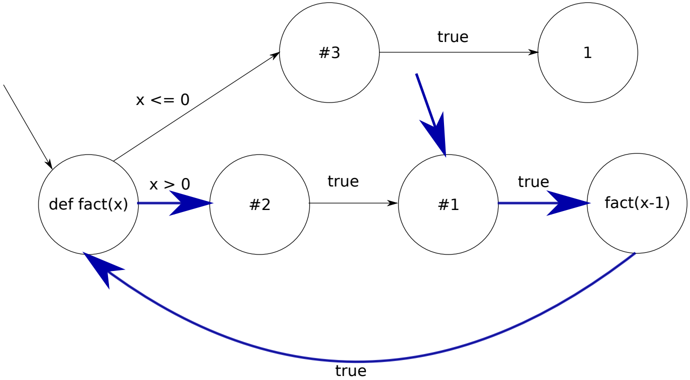


From fact to #1: $x > 0$ (Testcase:) $x = 1$

Generation of Deeper Testcases using Waypoints

Finding a path that visits each waypoint in order.

```
def fact(x: Int) = if(x <= 0) 1#3 else x*f(x-1)#1,#2
```

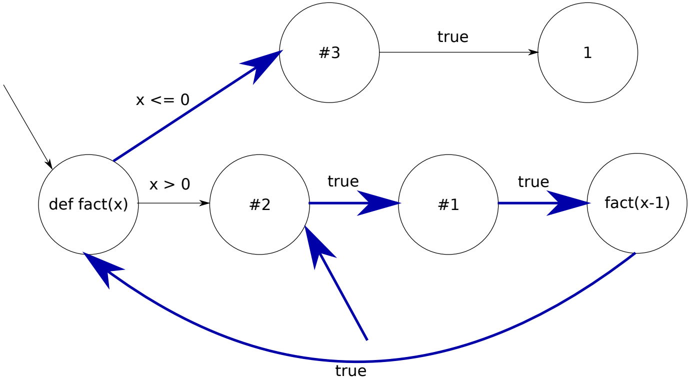


From `fact` to #2: $x > 0 \land x - 1 > 0$ (Testcase:) $x = 5$

Generation of Deeper Testcases using Waypoints

Finding a path that visits each waypoint in order.

```
def fact(x: Int) = if(x <= 0) 1#3 else x*f(x-1)#1,#2
```



From `fact` to #3: $x > 0 \wedge x - 1 > 0 \wedge x - 2 \leq 0$ Testcase: $x = 2$

Reasoning about Non-Deterministic Executions

```
def nonDeterministicExecution() {
  var i = 0
  var list: SortedList = Cons(42, Nil())
  repeat(2) {
    val b = epsilon((x: Boolean) => true)
    val n = epsilon((x: Int) => true)
    if(b)
      list = insert(list, n)
    else {
      list = remove(list, n)
      assert(!content(list).contains(n))
    }
  }
}
```

Reasoning about Non-Deterministic Executions

```
def nonDeterministicExecution() {
  var i = 0
  var list: SortedList = Cons(42, Nil())
  repeat(2) {
    val b = epsilon((x: Boolean) => true)
    val n = epsilon((x: Int) => true)
    if(b)
      list = insert(list, n)
    else {
      list = remove(list, n)
      assert(!content(list).contains(n))
    }
  }
}
```

$\mathtt{b}^1 = \mathtt{true}$, $\mathtt{n}^1 = 1691$, $\mathtt{b}^2 = \mathtt{false}$, $\mathtt{n}^2 = 42$

Reasoning about Non-Deterministic Executions

```scala
def nonDeterministicExecution(): Int = {
  var i = 0
  var b = epsilon((x: Boolean) => true)
  while(b) {
    i = i + 1
    b = epsilon((x: Boolean) => true)
  }
  i
} ensuring(_ <= 10)
```

Reasoning about Non-Deterministic Executions

```
def nonDeterministicExecution(): Int = {
  var i = 0
  var b = epsilon((x: Boolean) => true)
  while(b) {
    i = i + 1
    b = epsilon((x: Boolean) => true)
  }
  i
} ensuring(_ <= 10)
```

$\mathrm{b} = \texttt{if}(0 \leq \mathrm{i} \leq 10)\ \texttt{true else false}$

Epsilon Encoding


```
def foo(a: Int): Int = {
  epsilon(
    (i: Int) => i > 0
  )
}
```

Epsilon Encoding


```
def foo(a: Int): Int = {
  epsilon(
    (i: Int) => i > 0
  )
}
```

```
def foo(a: Int): Int = {
  def e1(): Int = {
    undefined
  } ensuring(res => res > 0)
  e1()
}
```

Epsilon Encoding


```
def foo(a: Int): Int = {
  epsilon(
    (i: Int) => i > 0
  )
}
```

```
def foo(a: Int): Int = {
  def e1(a1:  Int): Int = {
    undefined
  } ensuring(res => res > 0)
  e1(a)
}
```

Further Contributions

- Tuples added to the functional language, can be used via pattern matching.

```
val (x1, x2, x3) = (1, 2, 3)
```

- Support for functional and imperative `Array`, no aliasing.

```
a(i) = e
```

```
a = a.updated(i, e)
```

- Native `List` type and pattern matching.
- Various additional operations: modulo, `instanceOf` operator.

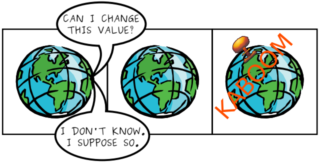
Overview of some Results

Benchmark	**LOC**	**#VCs**			**Time** (s)
		V	I	U	
`Arithmetic`	73	10	1	0	0.33
`ArrayOperations`	207	36	0	7	2.37
`ListOperations`	146	21	4	1	4.34
`Constraints`	76	6	3	1	2.41

- Each verification condition (VC) can be Valid, Invalid or Unknown (timeout).
- Different kinds of VCs:
  - loop invariants,
  - preconditions,
  - postconditions,
  - array accesses, and
  - exhaustiveness of match expressions.

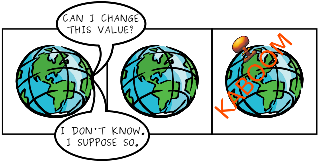
Limitations and Future Work

- Global variables (mutable fields).



Limitations and Future Work

- Global variables (mutable fields).

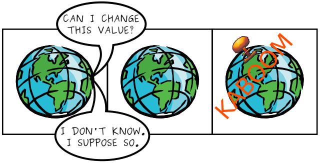


- Higher-order functions.

```
def map(lst: List, f: Int => Int): List = {
  //f is monotonic ?
  ...
}
```

Limitations and Future Work

- Global variables (mutable fields).



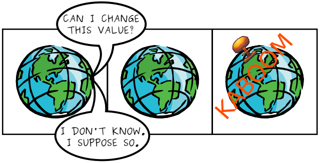
- Higher-order functions.

```
def map(lst: List, f: Int => Int): List = {
  //f is monotonic ?
  ...
}
```

- Induction and generalization.

Limitations and Future Work

- Global variables (mutable fields).



- Higher-order functions.

```
def map(lst: List, f: Int => Int): List = {
  //f is monotonic ?
  ...
}
```

- Induction and generalization.
- Loop invariant generation.

Related Work

- ACL2

  Kaufmann, Manolios, Moore, Computer-Aided Reasoning: An Approach, 2000

  - First-order Common Lisp.
  - Solving technologiy based on heuristics to apply rewrite rules.
  - Interactive, automatically generate subgoals.
- Imperative to Functional
  - Well-known result.
  - Recent work, formalized and proved in an interactive prover.

    Myreen, Formal verification of machine-code programs, PhD dissertation, 2008

Related Work

- Guardol

  Hardin, Whalen, Pham, The guardol language and verification system, TACAS 2012

  - DSL language to write and specify network guards.
  - Input language has imperative features that are mapped to functional equivalent.
  - Use an independant implementation of the same solving algorithm.

- HMC

  Jhala, Majumdar, Rybalchenko, HMC: Verifying Functional Programs Using Abstract Interpreters, CAV 2011

  - Translate functional programs into imperative programs.
  - Enable reuse of an interprocedural analysis for first-order imperative programs.

Conclusion

- Leon is now complete for finding counterexamples of *imperative* programs.
- Implementation of a general method to encode imperative programming into functional programming.
- Automatic generation of testcases implemented in Leon, including an advanced technique to test recursive functions.
- Additional datatypes and operations implemented in Leon.

Do you Have any Questions?

Please ask