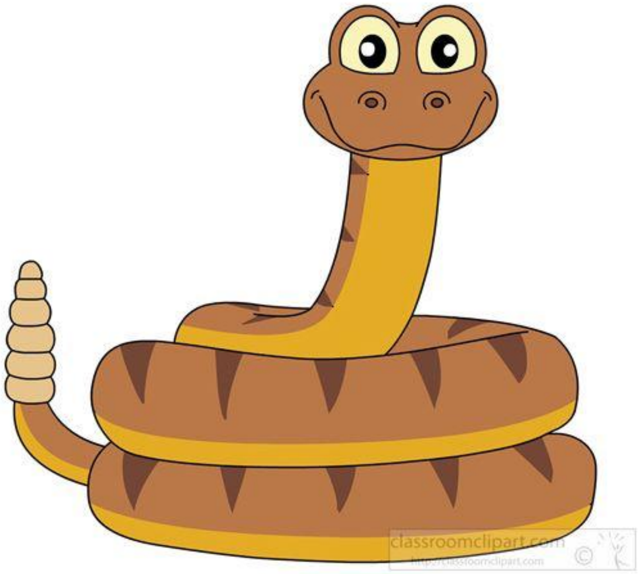

Verifying compiler for Rattlesnake

Valentin Aebi, Joseph Fourment, Luca Multazzu

Rattlesnake

Data types:

- `Int, Bool, Double, Char, Void, Nothing`
- `String`
- `arr Int`
- `struct Foo { bar : Int }`

Functions:

- `fn bar(i:Int) -> String { ... }`

If-Then-Else:

- `if < cond > { … } else if < cond > { ... } else { … }`

Loops:

- `while < cond > { … }`
- `for < stat > ; < cond > ; < stat > { … }`

Panic:

- `panic < message >`

Valentin Aebi, Joseph Fourment, Luca Multazzu

Assertions, Preconditions and Postconditions

Assertions:

- `assert < predicate >`

Preconditions:

- `require < predicate >`

Postconditions:

- `ensure < predicate >`

Loop Invariants

- `while < cond > invar < predicate >`

```
fn foo(x: Int, y: Int) -> Int
require x > 0
require y >= 0
{
    return 2*x + y
}
  ensure result > y
```

Valentin Aebi, Joseph Fourment, Luca Multazzu

Objectives

Add verification capabilities to Rattlesnake

- Preconditions, Postconditions, assertions and loop invariants
- Convert a program into formulas
- Feed the formula to a solver
- Output result (sat, unsat) and a readable explanation behind why something was unsat

Correctness

A function is **Partially Correct** if, provided:

- Its preconditions are satisfied upon entry
- The function returns

Then:

- Its postcondition is satisfied upon return

A function is **Totally Correct** if, provided:

- Its preconditions are satisfied upon entry

Then:

- The function returns
- Its postcondition is satisfied upon return

Valentin Aebi, Joseph Fourment, Luca Multazzu

Basic Paths

Sequence of statements $S_1; \ldots; S_n$:

- Starting with a precondition, loop invariant or assertion F
- Ending with a postcondition, loop invariant or assertion G

Represented as a Hoare Triple:

- $\{F\}S_1; \ldots; S_n\{G\}$

This triple encodes the **Verification Condition (VC)** of the path:

- Set of verification conditions is equal to the program formula

Valentin Aebi, Joseph Fourment, Luca Multazzu

In practice…

```
fn fib(n: Int) -> Int require n >= 0 {
    if n <= 1 {
        return n;
    };
    var prev = 0;
    var curr = 1;
    for var i = 2; i <= n; i += 1 invar curr > 0 invar prev >= 0 {
        val next = prev + curr;
        prev = curr;
        curr = next;
    };
    return curr;
} ensure result >= 0


fn main(args: arr String){
    print(intToString(fib(10)))
}
```

Desugaring


```
fn fib(n: Int) -> Int
require n >= 0 {
    if n <= 1 {
        return n;
    };
    var prev = 0;
    var curr = 1;
    for var i = 2; i <= n; i += 1
    invar curr > 0
    invar prev >= 0 {
        val next = prev + curr;
        prev = curr;
        curr = next;
    };
    return curr;
} ensure result >= 0


fn main(args: arr String){
    print(intToString(fib(10)))
}
```

```
fn fib(n: Int) -> Int {
  assume n >= 0;
  if n <= 1 {
   assume n <= 1;
    { val $0: Int = n;
      assert $0 >= 0;
      return $0   }
  } else assume !(n <= 1);
  var prev: Int = 0;
  var curr: Int = 1;
  { var i: Int = 2;
    { assert curr > 0;
      assert prev >= 0;
      while i <= n {
        assume i <= n;
        assume curr > 0;
        assume prev >= 0;
        val next: Int = prev + curr;
        prev = curr;
        curr = next;
        i = i + 1;
        assert curr > 0;
        assert prev >= 0 };
      assume !(i <= n);
      assume curr > 0;
      assume prev >= 0  }};
  { val $1: Int = curr;
   assert $1 >= 0;
   return $1   }}
```

```
fn main(args: arr String){
  print(intToString({
    val $2: Int = 10;
    assert $2 >= 0;
    val $3: Int = fib($2);
    assume $3 >= 0;
    $3
  }))
}
```

Path generation

Valentin Aebi, Joseph Fourment, Luca Multazzu

```
fn fib(n: Int) -> Int {
  assume n >= 0;
  if n <= 1 {
    assume n <= 1;
    {
      val $0: Int = n;
      assert $0 >= 0;
      return $0
    }
  } else assume !(n <= 1);
  var prev: Int = 0;
  var curr: Int = 1;
  {
    var i: Int = 2;
    {
      assert curr > 0;
      assert prev >= 0;
      while i <= n {
        assume i <= n;
        assume curr > 0;
         ...
```

Path generation

Valentin Aebi, Joseph Fourment, Luca Multazzu

```
fn fib(n: Int) -> Int {
  assume n >= 0;
  if n <= 1 {
    assume n <= 1;
    {
      val $0: Int = n;
      assert $0 >= 0;
      return $0
    }
  } else assume !(n <= 1);
  var prev: Int = 0;
  var curr: Int = 1;
  {
    var i: Int = 2;
    {
      assert curr > 0;
      assert prev >= 0;
      while i <= n {
        assume i <= n;
        assume curr > 0;
         ...
```

Path builder 1

Path generation

```
fn fib(n: Int) -> Int {
  assume n >= 0;
  if n <= 1 {
    assume n <= 1;
    {
      val $0: Int = n;
      assert $0 >= 0;
      return $0
    }
  } else assume !(n <= 1);
  var prev: Int = 0;
  var curr: Int = 1;
  {
    var i: Int = 2;
    {
      assert curr > 0;
      assert prev >= 0;
      while i <= n {
        assume i <= n;
        assume curr > 0;
         ...
```

Path builder 1

`assume n >= 0;`

Valentin Aebi, Joseph Fourment, Luca Multazzu

Path generation


```
fn fib(n: Int) -> Int {
  assume n >= 0;
  if n <= 1 {
    assume n <= 1;
    {
      val $0: Int = n;
      assert $0 >= 0;
      return $0
    }
  } else assume !(n <= 1);
  var prev: Int = 0;
  var curr: Int = 1;
  {
    var i: Int = 2;
    {
      assert curr > 0;
      assert prev >= 0;
      while i <= n {
        assume i <= n;
        assume curr > 0;
         ...
```

Path builder 1
`assume n >= 0;`

Path builder 2
`assume n >= 0;`

Valentin Aebi, Joseph Fourment, Luca Multazzu

Path generation


```
fn fib(n: Int) -> Int {
  assume n >= 0;
  if n <= 1 {
    assume n <= 1;
    {
      val $0: Int = n;
      assert $0 >= 0;
      return $0
    }
  } else assume !(n <= 1);
  var prev: Int = 0;
  var curr: Int = 1;
  {
    var i: Int = 2;
    {
      assert curr > 0;
      assert prev >= 0;
      while i <= n {
        assume i <= n;
        assume curr > 0;
         ...
```

Path builder 1
`assume n >= 0;`
`assume n <= 1;`

Path builder 2
`assume n >= 0;`
`assume !(n <= 1);`

Path generation

```
fn fib(n: Int) -> Int {
  assume n >= 0;
  if n <= 1 {
    assume n <= 1;
    {
      val $0: Int = n;
      assert $0 >= 0;
      return $0
    }
  } else assume !(n <= 1);
  var prev: Int = 0;
  var curr: Int = 1;
  {
    var i: Int = 2;
    {
      assert curr > 0;
      assert prev >= 0;
      while i <= n {
        assume i <= n;
        assume curr > 0;
         ...
```

Path builder 1	Path builder 2
`assume n >= 0;`	`assume n >= 0;`
`assume n <= 1;`	`assume !(n <= 1);`
`val $0: Int = n;`	`var prev: Int = 0;`

Valentin Aebi, Joseph Fourment, Luca Multazzu

Path generation


```
fn fib(n: Int) -> Int {
  assume n >= 0;
  if n <= 1 {
    assume n <= 1;
    {
      val $0: Int = n;
      assert $0 >= 0;
      return $0
    }
  } else assume !(n <= 1);
  var prev: Int = 0;
  var curr: Int = 1;
  {
    var i: Int = 2;
    {
      assert curr > 0;
      assert prev >= 0;
      while i <= n {
        assume i <= n;
        assume curr > 0;
         ...
```

Path builder 1

assume n >= 0;
assume n <= 1;
val $0: Int = n;
assume $0 >= 0;

Path builder 2

assume n >= 0;
assume !(n <= 1);
var prev: Int = 0;
var curr: Int = 1;

New path

```
assume n >= 0;
assume n <= 1;
val $0: Int = n;
-----------------
$0 >= 0
```

Valentin Aebi, Joseph Fourment, Luca Multazzu

Path generation

```
fn fib(n: Int) -> Int {
  assume n >= 0;
  if n <= 1 {
    assume n <= 1;
    {
      val $0: Int = n;
      assert $0 >= 0;
      return $0
    }
  } else assume !(n <= 1);
  var prev: Int = 0;
  var curr: Int = 1;
  {
    var i: Int = 2;
    {
      assert curr > 0;
      assert prev >= 0;
      while i <= n {
        assume i <= n;
        assume curr > 0;
         ...
```

Path builder 1

Dropped

Path builder 2

`assume n >= 0;`
`assume !(n <= 1);`
`var prev: Int = 0;`
`var curr: Int = 1;`
`var i: Int = 2;`

Formula generation

Path

```
assume n >= 0;
assume n <= 1;
val $0: Int = n;
----------------
$0 >= 0
```

Valentin Aebi, Joseph Fourment, Luca Multazzu

Formula generation

Path

```
assume n >= 0;
assume n <= 1;
val $0: Int = n;
----------------
$0 >= 0
```

$n \geq 0$ and $n \leq 1$ and $\$0 == n \Rightarrow \$0 \geq 0$
must be valid

Valentin Aebi, Joseph Fourment, Luca Multazzu

Formula generation

Path

```
assume n >= 0;
assume n <= 1;
val $0: Int = n;
-----------------
$0 >= 0
```

$$n \geq 0 \ \text{ and } \ n \leq 1 \ \text{ and } \ \$0 == n \ \Rightarrow \ \$0 \geq 0$$

must be valid

⇩

$$\neg(n \geq 0 \ \text{ and } \ n \leq 1 \ \text{ and } \ \$0 == n \ \Rightarrow \ \$0 \geq 0)$$

must be unsatisfiable

Valentin Aebi, Joseph Fourment, Luca Multazzu

Formulas to solver: SMT-LIB

- SMT-LIB is a standardized interface to SMT solvers
- Uses simple S-expr syntax to describe formulas and commands
- Example of SMT file:

```
(assert (not (forall ((x Int) (y Int)) (= x y))))
(assert (exists ((x Int) (y Int)) (= x y)))
(assert (=> (forall ((x Int) (y Int)) (= x y)) (exists ((x Int) (y Int)) (= x y))))
(check-sat)
```

Provided the set of clauses from a Rattlesnake program:

- Translate it to an SMT-LIB compatible file
- Check for satisfiability
- Call Z3 on said file and parse its output
- Return output

Valentin Aebi, Joseph Fourment, Luca Multazzu

Further work: faster SMT queries

- SMT-LIB provides a relatively high level interface to SMT solvers
- It allows us to store assertions in a stack
- We can exploit this to minimize the size of SMT queries and allow the solver to reuse its previous work:

```
(assert P)
(assert Q)
(check-sat)
(reset-assertions)
(assert P)
(assert Q)
(assert R)
(check-sat)
(reset-assertions)
(assert P)
(assert Q)
(assert R’)
(check-sat)
```

```
(push 1)
(assert P)
(assert Q)
(check-sat)
(push 1)
(assert R)
(check-sat) ; reuse P and Q
(pop 1)     ; only clear R
(assert R’) ; reuse P and Q
(check-sat)
(pop 1)
```

Further work: more types

- Add ability to handle arrays
  - Currently not supported
  - Extremely restricted unless we extend the class of predicates that we allow (e.g. inductive predicates)
  - Other solution: differentiate formulas from simple rattlesnake expressions of type bool
  - Cannot express interesting array predicates otherwise:

```
forall i, i <= 0 && i < array.length => array[i] = 0
```

- Doubles
- Chars & strings
- Structs

Further work: optional

- Use Inox
  - Currently we call Z3 on the SMT generated file
- Total Correctness
  - We only provide partial correctness
  - No checks on function termination

Conclusions

- This project aims at adding verification to Rattlesnake
- It converts programs to formulas by generating basic paths
- It feeds formulas to the Z3 solver
- It parses and returns the solver's output

References

1. Aaron Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification*, chapter 5. Program Correctness: Mechanics, pages 113–151. Springer Science & Business Media, 2007.
2. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*, www.SMT-LIB.org. 2016.

Desugaring

```
fn bar(x: Int) -> Int
require x >= 2
require x % 5 == 0
{
    var y = x;
    while (y < 48) invar y % 5 == 0 {
        y += 5;
    };
    return y;
} ensure result % 5 == 0

fn foo(){
    val abc = 20;
    print(intToString(bar(abc)))
}
```

```
fn bar(x: Int) -> Int {
  assume x >= 2;
  assume x % 5 == 0;
  var y: Int = x;
  {
    assert y % 5 == 0;
    while y < 48 {
      assume y < 48;
      assume y % 5 == 0;
      y = y + 5;
      assert y % 5 == 0
    };
    assume !(y < 48);
    assume y % 5 == 0
  };
  {
    val $0: Int = y;
    assert $0 % 5 == 0;
    return $0
  }}

fn foo(){
  val abc: Int = 20;
  print(intToString({
    val $1: Int = abc;
    assert $1 >= 2;
    assert $1 % 5 == 0;
    val $2: Int = bar($1);
    assume $2 % 5 == 0;
    $2
  }))}
```


Valentin Aebi, Joseph Fourment, Luca Multazzu