# Verifying compiler for Rattlesnake

Valentin Aebi
valentin.aebi@epfl.ch

Joseph Fourment
joseph.fourment@epfl.ch

Luca Multazzu
luca.multazzu@epfl.ch

January 2023

**Abstract**

This report presents a verifying compiler for the Rattlesnake programming language. The compiler is able to verify that programs using the *boolean* and *integer* data types obey specifications expressed as function pre- or post-conditions, loop invariants, assumptions or assertions. It makes use of the concept of basic path during the generation of verification conditions. Its limitations include the impossibility of verifying programs that use more complicated data types and the absence of termination checking.

## 1 Introduction

This project aims at adding verification capabilities to the Rattlesnake language, a statically typed, imperative language compiled to the Java Virtual Machine (JVM).

This project is based on the theory provided in Chapter 5 of *The calculus of computation: decision procedures with applications to verification* [3]. In particular, we make use of its notion of partial correctness defined as:

If

1. the precondition is satisfied upon entry
2. the function returns

then

the postcondition is satisfied upon return.

Furthermore, the way we build formulas for a program is inspired by (but not identical to) their description of *basic paths*. A basic path as explained in the book is a sequence of statements starting with a pre-condition and ending with a post-condition, such that if the precondition holds before the execution of the statements, then the post-condition should hold after their execution.

On the syntactic level, the project consists in augmenting Rattlesnake with statements that allow the verification of programs, namely assertions, loop invariants, assumptions, pre- and post-conditions. These statements are then verified by the verifier. The code is desugared, broken down into basic paths and transformed into logical formulas, which are written to an SMT file and verified by the Z3 solver.

With this work, we are now able to verify programs using formulas that involve the integer and/or boolean data types in Rattlesnake. We can also verify assertions on the length of arrays, in particular we can check that array accesses are within bounds.

## 2 The Rattlesnake programming language

### 2.1 Language core

Rattlesnake is a statically typed imperative toy programming language, initially developed as a personal project by Valentin. Its compiler targets JVM bytecode.

Rattlesnake supports a variety of types:

- Primitive Types: `Int`, `Bool`, `Char`, `Double`, `String`, `Void` (return type of a function that does not return anything), `Nothing` (return type of a function that never returns because it terminates the program)
- Arrays
- Structures

Regarding control structures, it supports functions, for and while loops, as well as if-then-else, return statements and a ternary operator that has the following syntax: `when <cond> then <expr> else <expr>`.

### 2.2 Verification statements

This project adds a number of extra statements to the already existing language, in order to provide verification capabilities. In particular, one may declare assertions and assumptions:

```
assert x >= 42
assume y < x
```

The difference between them is that the verifier will try to prove assertions, and report a failure if it cannot, while it will accept assumptions without trying to prove them.

Moreover, it is possible to provide pre-conditions before the body of a function using a `require` clause, and post-conditions after the body of a function using an `ensure` clause:

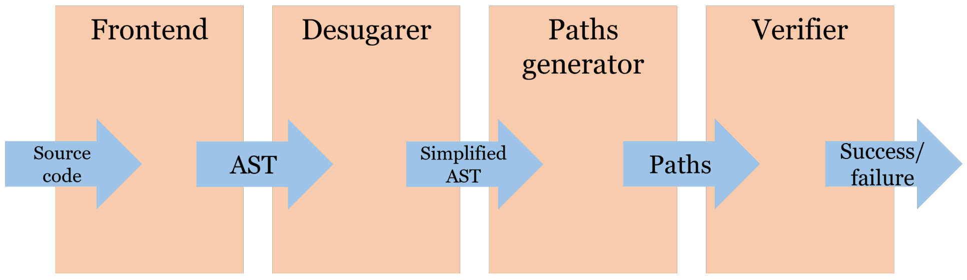
```
fn nonNegativeIdentity(x: Int) -> Int require x >= 0 {
    return x
} ensure result >= 0
```

Finally, loops admit loop invariants. The syntax is as follows:

```
while i < 10 invar i % 3 != 0 {
    ...
}
```

## 3 Implementation

The verifier is the composition of several steps. The first one is the front-end, which parses and typechecks the program. The other ones perform tasks that are specific to verification and will therefore be detailed in this section.



### 3.1 Desugaring

Desugaring has the following objectives:

- Reduce the set of language constructs that the subsequent steps of the pipeline (either the compiler or the verifier) have to handle. It does this by transforming some constructs into equivalent simpler ones. E.g. for loops are transformed into while loops;
- Ease the lazy evaluation of the `&&` and `||` operators by replacing them with occurrences of the ternary operator;
- Add assumptions that follow from the control flow of the program:
  - The *then* branch of an if or a ternary operator can assume that the condition of the if or ternary operator holds;

  - The *else* branch of the same construct can assume that the condition does not hold;
  - The body of a loop can assume that the loop condition holds at its beginning (otherwise the control-flow would have exited the loop);
  - Right after the loop, one can assume that the loop condition does not hold.

- Add assertions before every array access to make sure that no out-of-bounds error occurs, e.g. an occurrence of `x[i]` must satisfy `assert 0 <= i && i < #x`;

- Make the verification conditions (assertions and assumptions) explicit in the program. More precisely, it performs the following transformations:

  **Preconditions** are replaced by both an assertion at call site (to check the precondition on the actual values of the arguments) and an assumption at the beginning of the body of the function (since the condition is checked at call site, it can be assumed inside the function).

  Similarly, **postconditions** are desugared into both an assertion at each exit point of the function (to check that the postcondition holds on the value to be returned) and an assumption at call site, on the value returned by the function.

  For example, the `nonNegativeIdentity` function defined above gets desugared in the following way:

  ```
  fn nonNegativeIdentity(x: Int) -> Int {
    assume x >= 0;    // pre-condition
    {
      val $0: Int = x;
      assert $0 >= 0; // post-condition
      return $0
    }
  }
  ```

  A call to this function with argument 15 (`nonNegativeIdentity(15)`) gets desugared into the following:

```
val $1: Int = 15;
assert $1 >= 0;  // pre-condition
val $2: Int = nonNegativeIdentity($1);
assume $2 >= 0;  // post-condition
assume $2 == $1; // automatically added post-condition
... // do something with $2
```

Note that a second post-condition appeared at call site. This one was automatically inferred. Indeed, in the special case of functions whose body is `return <expr>`, the trivial postcondition `result == <expr>` is added by the desugarer.

Regarding **loop invariants**, we have to check both initialization and induction. For initialization, the invariant is asserted before the beginning of the loop. For induction, the invariant is assumed at the beginning of the body of the loop and asserted at the end of the body. This proves that if the invariant holds at the beginning of iteration *i*, then it also holds at the end of iteration *i*. Since the condition of the loop is not allowed to have side effects, this guarantees that the invariant also holds at the beginning of iteration *i+1*. Finally, the invariant is assumed after the loop, since it has been proven to hold after any number of iterations of the loop.

As an example, the loop defined above with loop invariant `i % 3 != 0` is transformed into the following:

```
assert i % 3 != 0;      // initialization
while i < 10 {
  assume i < 10;        // loop condition
  assume i % 3 != 0;    // induction: assumption
  ...
  assert i % 3 != 0;    // induction: assertion
};
assume !(i < 10);       // loop condition
assume i % 3 != 0       // proved by the invariant
```

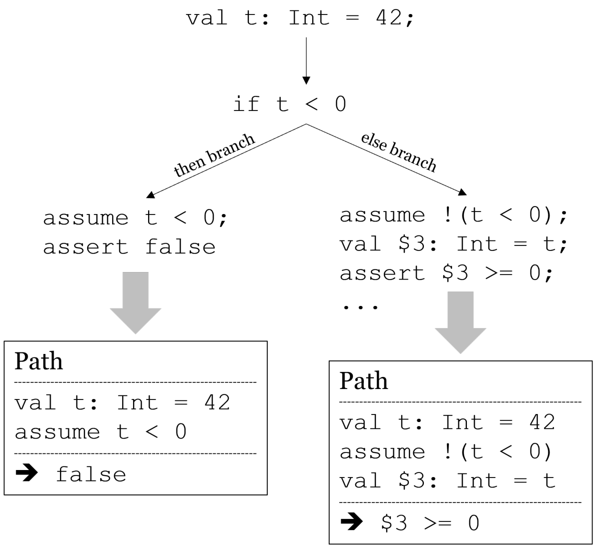
## 3.2 Paths generation

### 3.2.1 Control-flow analysis

Once desugared, the program is ready to be transformed into paths. To do so, the `PathsGenerator` step of the verifier analyzes the control-flow of the program. It uses `PathBuilders`, which can be seen as mutable lists of statements that follow each possible control-flow path. A path builder is first initialized at the beginning of each function, and then statements are added to it in their order of execution in the program. When an assertion is encountered, the path builder generates a new path, that is, a sequence of statements at the end of which the asserted formula must be true [1]. The same path builder will then continue its analysis of the control-flow. The assertion is added to the path builder *as an assumption*. Indeed, since a path has been created to check the validity of the assertion, the subsequent assertions can be proven using that one.

When the control-flow analysis encounters an **if**, the path builders get duplicated, so that both branches can be analyzed. When it reaches a **loop**, the path builders are dropped (at that point they have already generated paths to prove the initialization step of the invariants). New empty path builders are created to analyze both the body of the loop and the statements after the loop. These new path builders initially do not contain any statement, therefore their only knowledge of the program is the invariants, which are assumed both at the beginning of the body and right after the loop, and the control-flow assumptions coming from the loop condition. This means that any formula proven before the loop that is needed to prove the correctness of another formula after the loop must be a loop invariant. Finally when a path builder reaches a **return**, it stops existing because this is the end of the control-flow branch under analysis.

[1]Note that this definition of a path slightly differs from the one given in [3]. Indeed, in our project paths have no pre-condition, but contain all the statements that happen before the asserted formula in the control flow, even if there is an assertion among these statements. The goal of this design is to make it possible for the user to omit the previously asserted formulas. Indeed, with the design of the book, a path only assumes its precondition, which means that the user must explicitely include all previously asserted formulas in each assertion to propagate them across the program.



### 3.2.2 Making paths functional

Before transforming these paths into formulas, we need to remove variable reassignments from them. Indeed, a reassignment means that a symbol (the name of the variable) does not have the same value at all points of the path. Since in a logical formula a variable cannot have more than one value, we need to transform these reassignments into definitions of new values.

For example,

```
var j = 101;
j += 29;
j = square(j)
```

becomes

```
val j = 101;
val j1 = j + 29;
val j2 = square(j1)
```

(actual identifiers after transformation are slightly more complicated)

## 3.3 Formula generation

Once processed, statements in paths are converted into logical formulas. Since we use the Scala SMT library [2], these formulas are represented using the types defined in this library. Three types of statements need to be converted:

- Definitions of locals, which are converted into a formula of the form *id = value*, e.g. `val j = 101` becomes *j = 101*;
- Assumptions, e.g. `assume i > 0`, which are already formulas (the assumed formula is simply translated into a Scala SMT-Lib formula);

- Assignments to arrays or struct fields. The compiler does not support the verification of such constructs, and thus generates a formula of the kind $w = lhs$ where $w$ is a variable that is used nowhere else and $lhs$ is the assigned value (left-hand side of the assignment), so that this formula does not impact the verification of the program.

All the formulas obtained from the statements of a path are then combined into a conjunction. We denote $s_1, ..., s_n$ the formulas obtained from the $n$ statements of a path and $p$ the postcondition at the end of the path. The path obeys its specification iff $a \wedge (\bigwedge_{k=1}^{n} s_k) \rightarrow p$, where $a$ is the following tautology on the length of arrays: $\forall x\, length(x) \geq 0$.

We need to feed this formula to a solver to verify its validity. However, solvers check satisfiability, not validity. Therefore, the verifier negates the formula before writing it to the SMT file [1], and the program will be considered correct iff the checker finds that the negated formula is unsatisfiable.

### 3.4 Solver invocation and reporting

After writing the formulas to a file, the verifier invokes the Z3 solver [4]. The default timeout is 2 seconds, but the user of the compiler can specify another value. This is done by running the appropriate command in a subprocess. The output of Z3 is then parsed. The possible results are:

- Sat[isfiable]: the program does not obey the specification;
- Unsat[isfiable]: the program obeys the specification;
- Timeout: the solver could not verify the formula in the allocated time;
- Error: if something unexpected happens, typically if Z3 is not installed on the user's computer.

If the result is Sat, then the counter-example found by Z3 (an assignment of the variables that satisfies the negated formula) is parsed, and the values assigned to variables visible to the user (i.e. not the ones generated by the compiler) are displayed to the user as a hint of what violation it was not able to prove impossible.

## 4 Further Work

### 4.1 Support for more data types

Currently, our project supports programs containing integers and/or booleans, as well as predicates on the size of arrays. However, it is possible to expand this set. Doubles can be supported by making use of the theory for floats already present in SMT-Lib and supported by most solvers. The most interesting types to handle are arrays and structures. We may allow for new predicates such as universal and existential quantifiers to be used in formulas as a way to expand the support for arrays, the goal being to express properties like "for all elements of the array". SMT-Lib's support for algebraic data types could also be used to express properties on structs. However, the fact that arrays and structs are mutable makes it difficult to verify their state.

### 4.2 Total Correctness

In this project, we define the correctness of a function with the notion of partial correctness. However, there is a more complete notion, known as *Total Correctness* [3]. This states that provided the pre-conditions of a function are satisfied upon entry, then the function returns, and its post-conditions are satisfied upon return. In other words, it coincides with partial correctness alongside proof of function termination. A possible, expansion of this work is to provide this further notion of correctness.

### 4.3 Technical improvements

Some parts of our implementation are not optimal. For example, the generation of formulas happens after the paths have been built, which means that statements that occur in several paths are translated into SMT-lib formulas more than once. An improvement would be to directly compute this transformation when the statements are added to the path builders. Furthermore, during path generation, the verifier duplicates all paths each time an if statement is encountered. This leads to an exponential growth of the number of builders, which could probably be avoided by merging the paths corresponding to both branches (then and else) into a single formula at the end of each if statement: $(condition \wedge thenPath) \vee (\neg condition \wedge elsePath)$.

## 5 Conclusion

This project has made it possible to verify simple properties of Rattlesnake code. While our verification system is unable to express complex specifications on complex programs, it is capable of verifying the partial correctness of small examples and demonstrates the basic techniques of the verification of computer programs.

Link to the GitHub repo of the verifying compiler:
https://github.com/FV-Rattlesnake-team/Rattlesnake

## References

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[2] Régis Blanc and Viktor Kuncak. Sound reasoning about integral data types with a reusable smt solver interface. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, SCALA 2015, page 35–40, New York, NY, USA, 2015. Association for Computing Machinery.

[3] Aaron R 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.

[4] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.