Rust2Viper: Building a static verifier for Rust

Florian Hahn

November 10, 2015

1 Context

Rust is a modern general purpose programming language with the goal of being "safe, concurrent and practical". The language comes with a powerful type system based on the concepts of ownership (including transfer of ownership) and borrowing. Rust's ownership system ensures that there is exactly one owner of any given resource. The owner of a resource can change by *moving* the resource from one binding to another. In addition, Rust allows the sharing of resources via references. The owner of a resource can hand out references to the resource, e.g. to pass it to a function as a parameter. The receiver of the references is said to borrow the resource, e.g. until the function returns in the case that the reference is passed as a function parameter. Furthermore Rust makes a distinction between mutable and immutable borrows. At any give time, a resource can be immutably borrowed one or more times or mutably borrowed exactly once, but not both.

Silver is an intermediate language with a flexible notion of permissions built in. It provides basic features such as methods, loops and conditionals and also provides a way to specify contracts using pre- and postconditions, loop invariants and assertions. A verifier for a high-level language can use the basic features of Silver to encode complex language constructs. After a program has been translated to Silver the Viper tool chain can be used to verify it using different back-ends [3]. At the moment there already exist front-ends for some programming languages, e.g. Chalice2Silver for Chalice and Scala2Silver for Scala [1].

A major difference between the existing front-ends and a front-end for Rust is that the Rust type system already provides information about the scope and mutability of bindings, which can be used to automatically infer permission specifications in Silver.

The main goal of this project is to implement a front-end which allows the verification of Rust programs using the Viper infrastructure.

2 Core Goals

This project focuses on a subset of Rust, which should be powerful enough to write meaningful programs, while being small enough to be implemented in the available time. This subset includes

- data structures (*struct*)
- enumerations (*enum*)
- basic control structures (*if, while, let*)
- functions (*fn*)
- pattern matching (*match*)
- implementation of structs and enums (*impl*)
- basic support for some data types (e.g. *Vec<T>*, including specification for functions such as *Vec.len()*)
- basic support for ownership, borrows and *Box<T>* pointers.

With this subset, it should be possible to verify a reasonably wide range of programs. As part of the core goals we do not plan to support converting Rust functions to Silver functions or predicates, but there is an extension goal which tackles this problem. In order to circumvent this initial limitation and verify interesting properties, we provide the user with a mechanism to specify functions and predicates as Silver code and use them in the contracts of the Rust program. In a similar fashion, users can provide specification for library Rust functions (e.g. *Vec.len()*)

The listings at the end of the document illustrate the supported features using three examples. The first example shows how to verify a saturating counter and illustrates almost all the syntax supported as part of the core goals. The second example shows a verified implementation of binary search and the third example shows how to verify properties of recursive data structures. Note that the *#[pure_function]* annotations in the third example are the only feature that is not supported as part of the core goals. To verify this example without the respective extension goal, Silver definitions for *itemAt()* and *length()* would have to be provided.

The examples are inspired by real world Rust code; the binary search implementation in the Rust standard library and the specification in [7], provided the foundation for the second example, while the third example was inspired by [6] and [2].

For specifying contracts, we want to use the same syntax as *libhoare*, a syntax extension for specifying function contracts, which are checked dynamically [4]. This way, it should be possible to verify some code statically that uses *libhoare*'s dynamic checks at the moment. However, we want to extend *libhoare*'s syntax to allow programmers to express more powerful specification, e.g. by using quantifiers or simple pattern matching in contracts.

The tool will be implemented in Rust, because the Rust compiler allows reusing infrastructure such as the parser and type checker. This means that the tool will not be able to generate a Silver AST directly and instead has to emit Silver code as text. Therefore special care has to be taken in order to map Silver verification errors back to the Rust source code.

3 Extensions

Improve verification error messages using back-tracking When the Rust compiler detects an error it tries hard to present a helpful error message by including helpful context (e.g. when the same data is borrowed as mutable twice it also displays the location where the first mutable borrow occurs). Therefore Rust programmers probably have high expectations with regard to error reporting. In the scope of this extension, we would look into ways to provide better error messages if the verification fails, e.g. by displaying the last location where a condition that is expected to hold later on was satisfied. This would require extending the verifier (*Silicon* or *Carbon*) to back-track on a verification error and inspect previous verifier states to find the cause of the error.

Pureness Checking This extension allows programmers to mark Rust functions as pure with an attribute *#[pure_function]*. These pure Rust functions are then translated to Silver functions, which can be used in contracts. As a first step, we just assume all functions marked as pure by the programmer are indeed pure and used correctly in contracts. As a second step, we want to check if functions marked as pure are well-formed, along the lines suggested in [5]. In particular we want to check whether the functions are indeed pure and if they are well-defined by analyzing the dependency graph of the used pure functions.

Handling unsafe blocks While Rust provides guarantees for normal Rust code (e.g. at any given time, at most one mutable reference can point to any given data structure), in *unsafe* blocks arbitrary pointer operations are allowed, which could be used to violate the guarantees after the *unsafe* block. This could lead to undefined behavior and unexpected errors. This extension goal consists of two steps. First we want to add basic support for *unsafe* blocks. At this stage, we assume that none of the guarantees is violated at the end of the *unsafe* block, e.g. no reference contains a null value, and treat the whole block as a black box. This means we forget all information about the values of bindings used in the *unsafe* block and the user has to provide unchecked assumptions that reflect what is going on in the block. Second we want to generate Silver code for *unsafe* blocks. Supporting every possible operation in *unsafe* blocks is out of scope of this extension. Instead we want to focus on *unsafe* blocks that deal only with pointer operations and memory allocations. Finally we want to verify some guarantees that should still hold after an *unsafe* block, like all references still point to non-null values or that there is at most on mutable reference to every resource, even though these properties might turn out too hard to verify in the general case.

Support for iterators and closures When dealing with sequential data structures, most prominently Rust's *Vec*, iterators play an important role. For example, Rust's *for* loop construct is just syntactic sugar for a looping over an iterator. Iterators also provide functions like *map()*, *filter()* or *fold()* that take closures as parameters. Using those functions when possible instead of *for* loops is considered good practice in the Rust community. Due to their widespread use, supporting *for* loops and iterator functions with closures would substantially increase the number of interesting programs the verifier can handle.

The aim of this extension goal is to add support for *for* loops and some of the iterator functions mentioned above. First we want to add support for *for* loops that iterate over a vector, by handling the loops before they are de-sugared. Then we want to look into ways of implementing basic support for closures, which could be used with iterator functions.

References

[1] Bernhard Brodowsky. "Scala to SIL". Master's Thesis. ETH Zurich, 2012.

[2] *ixlist — simpl doubly-linked list*. URL: `https://github.com/bluss/ixlist` (visited on 10/08/2015).

[3] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. *Viper: A Verification Infrastructure for Permission-Based Reasoning*. Tech. rep. ETH Zurich, 2014.

[4] *libhoare — Design by contract style assertions for Rust*. URL: `https://github.com/nrc/libhoare` (visited on 10/13/2015).

[5] Arsenii Rudich, Ádám Darvas, and Peter Müller. *Checking well-formedness of pure-method specifications*. Springer, 2008.

[6] *Rust by Example - Enums*. URL: `https://japaric.github.io/rbe/enum.html` (visited on 10/08/2015).

[7] *silver test suite- binarySearchSeq.sil*. URL: `https://bitbucket.org/viperproject/silver/src/18254eeb64353043ef9da6668c7a274b6c5ba090/src/test/resources/all/sequences/binarySearchSeq.sil` (visited on 10/28/2015).

```
enum CounterState {
  NotSaturated(i32),
  Saturated,
}

impl CounterState {
  #[precond="NotSaturated(v)== self"]
  fn unwrap(&self) -> i32 {
      match self {
          NotSaturated(v) => v,
          Saturated => panic!(),
      }
  }
}

struct SaturatingCounter {
  v: CounterState
}

impl SaturatingCounter {
  #[postcond="NotSaturated(v1) == old(self.v) &&
      v1 + v < 256 ==>
          NotSaturated(v1) == self.v && v1 < 256"]
  #[postcond="NotSaturated(v1) == old(self.v) &&
      v1 + v >= 256 ==>
          Saturated == self.v"]
  #[postcond="Saturated == old(self.v) ==>
      Saturated == self.v"]
  pub fn add(&mut self, v: i32) {
      match self.v {
          NotSaturated(v1) if v1 + v < 256 =>
              self.v = NotSaturated(v1 + v),
          _ => self.v = Saturated
      };

}

fn main() {
  let mut c1 = SaturatingCounter{v:NotSaturated(10)};

  c1.add(100);
  // OK
  c1.v.unwrap();

  let mut c2 = SaturatingCounter{v:NotSaturated(100)};
  c2.add(300);
  // not OK
  c2.v.unwrap();
}
```

Example 1 – Saturating counter

```
#[precond="forall i,j in [0..vec.len()-1] .
  i < j ==> vec[i] < vec[j]"]
#[postcond="result == Some(index) ==>
  index < vec.len() && vec[result] == key"]
#[postcond="result == None ==>
  (forall i i in [0..vec.len()-1] .array[i] != key)"]
fn binary_search(vec: &Vec<i32>, key: i32)
  -> Option<usize> {
    let mut low: usize = 0;
    let mut high: usize = vec.len();
    let mut mid;
    let mut index = None;
    while low < high {
    invariant!("0 <= low && low <= high && high <= |array|")
    invariant!("forall i in [0..vec.len()-1] .
            !(low <= i && i < high)) ==> vec[i] != key")
        mid = (low + high) / 2;
        if vec[mid] < key {
              low = mid + 1;
        } else if key < vec[mid] {
             high = mid;
        } else {
            index = Some(mid);
        }
    }
    index
}
```

Example 2 – Binary search

```
pub struct Cell {
    pub v: i32
}

pub enum List {
    Node(Cell, Box<List>),
    Nil
}

#[postcond="forall i in [0..length(list)-1] .
    j != 0 ==> itemAt(j) == old(itemAt(j)) + v"]
pub fn add(l: &mut List, v: i32) {
    match *l {
        List::Node(ref mut c, ref mut t) => {
            c.v = c.v + v;
            add(t, v);
        }
        List::Nil => ()
    };
}

#[pure_function]
#[postcond="result >= 0"]
pub fn length(l: &List) -> u32 {
    match *l {
        List::Node(_, ref tail) => 1 + length(tail),
        List::Nil => 0
    }
}

#[pure_function]
#[precond="i >= 0 && i < length(l)"]
pub fn itemAt(l: &List, i: u32) -> i32 {
    match *l {
        List::Node(ref c, _) if i == 0 => c.v,
        List::Node(_, ref tail) => itemAt(tail, i-1),
        List::Nil => panic!()
    }
}

#[postcond="length(list) == old(length(list)) + 1"]
#[postcond="itemAt(0) == c.v"]
#[postcond="forall i in [1..length(list)-1] .i
    j != 0 ==> itemAt(j) == old(itemAt(j))"]
pub fn append(list: List, c: Cell) -> List {
    List::Node(c, Box::new(list))
}
```

Example 3 – Recursive list implementation