Translating Rust into ML for proving Program Equivalence by Bisimulation

Edvinas Teišerskis

August 7, 2021

Abstract

I present a potential example of the Hobbit tool (Higher Order Bounded BIsimulation Tool) by compiling a subset of the Rust programming language into the ML-like language used by Hobbit. When realised, this compilation can serve as a method of comparing practical code segments for program equivalence. This was done by taking a subset of Rust and proceeding to create a mapping and compilation tool, named \texttt{rust2bils} into the verification language. Examples and benchmarks within the compiler tool subset that can be applied to the tool were tested and verified with the Hobbit tool, showing that it succeeds in compiling semi-practical code segments. I also sketched out how the compiler tool subset can be extended with future work, even to a state where it can potentially be applied to more practical systems.

1 Introduction & Background

In "There and Back Again: From Bounded Checking to Verification of Program Equivalence via Symbolic Up-to Techniques"\cite{2}, Koutavas et al. present a novel technique for verifying equivalence between programs, in addition with a tool prototype called Hobbit (Higher Order Bounded BIsimulation Tool) which implements the verification technique.

In this project I aim to show a practical usage of the tool, by constructing a compiler from a subset of a common, practical programming language, in this case the Rust programming language, to the verification language of the Hobbit tool, called 'bils'. As such, using the compiler and the Hobbit tool together, I aim to implement a system for comparing two, native and potentially practical programming language segments for equivalence.

In order to do so, I develop a potential mapping from the Rust subset into bils, in such a manner that the translation is equivalent in its meaning, and then I implement a compiler that reads Rust code, and proceeds to automatically apply the mapping in a practical manner.

The project resulted in a usable compiler tool, named \texttt{rust2bils}\footnote{found at \url{https://github.com/teiserse/rust2bils}} which I
benchmarked and tested with various examples of the Rust subset I aimed to implement, and proceed to test them with the Hobbit tool. The implemented subset contains enough features in order to successfully compile some basic algorithms. I also lay out some of the avenues in which the tool can be improved and the supported Rust subset can be expanded in order to expand the potential use cases of the compiler tool.

1.1 Formal Verification & Contextual Equivalence

Formal verification is the process and methods of automatically proving the correctness, or detecting incorrectness, in a system or an algorithm. This is separate from techniques like testing, where instead of running a particular system/algorithmpiece of code and checking for errors, the code is instead proved correct or incorrect to a certain condition (such as a model of the algorithm, or some necessary final requirement) using formal logic. These techniques are often used with low-level, systems programming code, where there is a very low tolerance of errors, or algorithms and protocols in fields such as cryptography, where formal correctness is highly desired.

The particular area of formal verification that this project is dealing with is contextual equivalence. The paper which introduces the Hobbit tool defines contextual equivalence as "a relation over program expressions which guarantees that related expressions are interchangeable in any program context." This effectively means that the system attempts to verify if a particular code expression is correct in regards to a target code expression, or put more simply, checking if two code expressions end up with the same, equivalent resulting state.

An example usage of this particular technique would be in the case of making an update to a function which is an implementation of an algorithm. In this case, some of the function is rearranged, or operators are changed and shifted, for a purpose like making the function more optimised. As such, in order to ensure that the correct algorithm is still maintained by the changes, a verifier system could take in the updated version, and check it with the original, unchanged implementation. If this check can determine equivalency, then the update writers can be certain that their changes have maintained the algorithm, or if a difference is found, they can catch the error before it goes into compilation and usage.

1.2 The Rust Programming Language

Rust is an imperative programming language, most notably characterised by its ownership-based type system. This system allows the language memory safe, while not using garbage collection. Instead, the language applies rules to how values are passed, used and referenced at compile time. A value has a unique owner scope, and must either be used by the owner scope, be borrowed by another scope via a reference, or have its ownership passed to a different scope.
As part of this system, owned values a dropped at the end of a scope, and cannot be used in any manner after the value is dropped.

Rust has also been growing in popularity and usage since its stable release in 2014. As per the PYPL Popularity of Programming Language Index [5], Rust has been consistently growing or stable in popularity since 2015, with even better metrics from the release of the 2018 edition of Rust in December 2018. In addition, it has attained a good reputation with developers, as it has been consistently voted the most loved language in Stack Overflow’s Annual Developer Survey [4]. Rust’s features also are making it very well regarded as a systems programming language, especially due to its memory safety [3]. The language has also attracted work with formal verification, with Jung et al. [1] presenting a formal proof for the safety of “a realistic subset” of Rust’s type system.

The previous paragraphs point to my reasoning to choose Rust as the source language for this project. Rust’s systems and guarantees mix in well with other formal verification methods, as the already existing guaranteed can be mixed in with additional verification systems for a more overall safe programming package. The types of projects that would be looking for these guarantees would also be likely to approach additional methods with which they can be more certain of the soundness of their work. Its rising popularity and acceptance also show that Rust is not just a research project or a niche subject, but a technology that already is widely used in practice. As such, working on projects such as this on Rust are more likely to touch on a wider variety of circumstances, and could be even applied to currently existing projects.

## 2 Language Mapping

The Hobbit tool checks equivalence between two expressions. The expressions checked with the tool are typically in the form of functions. As such, the goal of the mapping is to take a chosen function in Rust that we wish to check, along with other supporting items (typically other functions used by the main function), and map it into the equivalent bils syntax as a single, checkable expression.

Our mapping strategy is to take the format that the Rust code is presented as, and try to as closely match it within bils. Some of this process is helped by bils supporting sequencing expressions, which means that the project can work within the imperative programming framework. As such, I try to map as much of the concepts as I can more closely to how they appear in Rust. However, this does require us to perform extra work to make sure this format is preserved.

### 2.1 Feature mapping

Both Rust and bils share the typical set of arithmetic and logical operators, along with the semicolon syntax for sequenced operations. As such, their
theoretical mapping is trivial, as they are identical in both cases, outside of differences in symbols and arrangement.

2.1.1 Definitions

There are two types of definitions in bils - expressions, defined with `let` and locations, defined with `ref`. `let` definitions are immutable, while `ref` definitions are mutable. As such, for the purpose of compiling from Rust to bils, I map variables as `ref` definitions, since this allows to properly map how mutable variables are used in imperative languages. However, this requires the compiler to add boilerplate to definitions, in order to have the variables as `ref` definitions.

```plaintext
1 let xy = (1, 2) (* starting variables, e.g. to a function *)
2 let (x__prep, y__prep) = xy in (* tuple unpacking *)
3 ref x = x__prep in (* x == 1 *)
4 ref y = y__prep in
5 x := !x + !y; (* x == 3 *)
6 !x (* expression returns 3 *)
```

Listing 1: bils let/ref example

2.1.2 Functions

In bils, functions are defined either with the `fun` or the `rec` (recursive) keyword. Either case only takes one parameter. As such, in cases where we want to map multi-parameter functions, we can either curry multiple functions, or pass all the parameters as a single tuple, and then proceed to unpack the tuple at the beginning of the function. I decided to go with the latter, to more closely match Rust (which does not have explicit currying), and to allow easier working with the `rec` keyword.

```plaintext
1 fn add (x: i32 ,
2 y: i32 ) -> i32 {
3 x + y
4 }
```

Listing 2: Rust function example

```plaintext
1 (* recursive syntax *)
2 let rec increment params = (
3 (* fun keyword syntax *)
4 let increment = fun params -> (
5 let ( x__prep , y__prep ) = params in
6 ref x = x__prep in
7 ref y = y__prep in
8 x + y
9 ) in
```

Listing 3: bils mapped function examples

2.1.3 Code Structure

Code layout and structure is largely trivial, but can end up being incompatible in certain states of dependency. In Rust, the locations and ordering of items, such as functions, type definitions, and so on, are just organisational tools. In the case of things like a program entry point, they are either set by language defaults (`fn main()`) in the main/first file given to the compiler) or explicitly by the programmer. Therefore, they can be rearranged at will as long as relevant
identifiers are changed as needed. However, since Hobbit checks between two expressions, a list of declarations in bils has to be nested, so that the end result is a single expression. As such, items are defined as existing within a provided context, as opposed to universally.

For example, take a function \( A \), that calls function \( B \), which itself calls function \( C \). In Rust organisation, \( A \), \( B \) and \( C \) can appear in any ordering, as long as the proper location names and visibility keywords are used. However, bils requires us to explicitly define the items in sequential order, as \( A \) requires \( B \) to be in context in order to call it, with the same for \( B \) calling \( C \).

```rust
fn A() {
    B();
}

fn C() {
    // ...
}

fn B() {
    C();
}
```

Listing 4: Rust ordering example

```bils
let rec C params = ( (* ... *)
) in

let rec B params = ( C ()
) in

let A = fun params -> ( B ()
)
```

Listing 5: bils ordering example

bils has support for recursive functions, as noted with the `rec` keyword, but still within the limits of the given context. While we are able to define a function that calls itself (function \( A \) that calls function \( A \)), We can end up in scenarios of mutual recursion (function \( A \) that calls function \( B \), while \( B \) calls \( A \)) where the context does not allow such functions to be defined. In such a case, potentially mapping Rust to bils would require changes that could change the meaning of the program, such as wrapping the functions in a secondary function in order to work with the bils contexts.

```rust
fn a() {
    // ...
    b();
    // ...
}

fn b() {
    // ...
    a();
    // ...
}
```

Listing 6: Rust mutual recursion

```bils
let a = fun params -> ( ... b ... ) in

let b = fun params -> ( ... a ... ) in
```

Listing 7: bils - incorrect, causes errors

```bils
let ab = fun selection -> ( if ( selection == 0 ) then ( (* a *)
) else ( (* b *)
) ) in
```

Listing 8: bils - compiles, but redefined

### 2.1.4 Control Flow

Common code concepts for control flow maps mostly directly. Conditionals and sequencing have equivalents in bils, however looping does not. As such, I decided to map simple while loops as a local recursive function, which contains a conditional containing the inner code of the loop, and then calling the recursive function at the location of the loop:
2.1.5 Type Primitives

For primitives, bils supports signed integers, booleans and blank ”unit” types. The corresponding types in Rust consist of its primitive integer types (which are explicitly sized and signed/unsigned) booleans and unit structs. However, there can be some issue when comparing Rust integers with bils integers. The integers in Hobbit are unbounded. This can cause issues in cases where the size of the integer in Rust can be relevant. As an example, take the smallest signed integers in Rust, i8, which can range from -128 to 127.

```rust
let num1: i8 = 100;
let num2: i8 = 100;
let num3 = num1 + num2; // overflow!
```

In the above case, Rust code will either end up with a value of -56, or will raise an error depending on compilation settings. However, due to Hobbit’s unbounded integers, these types of errors are completely ignored by the system. Due to this, the compiler has an additional requirement that any program checked with the system will have no issues, conflicts or operations based on over/underflow of integers.

2.1.6 Compound Types

In terms of compound data types, such as tuples, arrays and structs, I map them all as bils tuples. bils already uses tuples and arrays interchangeably, and structs can be represented as a tuple of its items in the struct’s ordering. This method does not preserve the type system in this regard in bils (e.g. two different struct types with the same field types and orderings will be represented identically), but as long as the Rust compiler’s type system check holds, we can assume that there will be no type errors.
In the case of arrays, there is also the matter of syntactic sugar in order to accommodate for array value getting and setting by index. While this can be easily ignored in the matter of read-only arrays and simply be unpacked as tuples, this accommodates mutable, settable arrays.

```rust
let array = (1, 2, 3, 4);

let array__get i = if (i < 0 || i >= 4) then _bot_ else
    if i = 0 then !array[0/4] else if i = 1 then !array[1/4] else
    if i = 2 then !array[2/4] else if i = 3 then !array[3/4] else
    _bot_ in

let array__set i = fun m -> if (i < 0 || i >= 4) then _bot_ else
    if i = 0 then array := !array[0/4:= m] else
    if i = 1 then array := !array[1/4:= m] else
    if i = 2 then array := !array[2/4:= m] else
    if i = 3 then array := !array[3/4:= m] else
    _bot_ in
```

Listing 14: Array syntactic sugar for bils example

2.1.7 References

One of the concepts that I map which is more specific to Rust, as it is related to the ownership system, is references. Since a variable that someone creates a reference to is still owned by the scope which created the reference, the mapping needs to keep the variable within the scope and instead provide a way that this variable would be accessed. This is particularly important in the case of changing the variable’s value, as it will need to be changed in the owner’s scope, and not just locally. In compiled Rust, this would act as a pointer, but to keep it within the abstractions of bils the compiler instead creates a getter and setter function pair, which takes the pointer’s place.
### 2.2 Rust’s Ownership Type System

In this process, there also has to be the consideration of the Rust ownership type system. However, this can be easily abstracted out, with the condition that whatever Rust code in being checked is valid and compilable Rust code. As the ownership type system guarantees memory and type safety, a program that is compiled can be certain of those guarantees, and as such, these checks generally do not need to be considered.

One example of this can be Rust’s distinction between mutable and immutable variables. In Rust, variables are immutable by default, and must be declared mutable using the `mut` keyword. This has a reflection in the bijls definition format, where there is a difference between using `let` (immutable) and `ref` (mutable). While it can be reasonable to maintain Rust’s distinction between mutable and immutable variables, by defining mutable variables using `ref`, and immutable variables using `let`, it’s unnecessary, and I avoid it to keep the mapping simpler. The improper usage of (im)mutable variables is already included in Rust’s type system check, and there is no change in meaning if a variable is declared mutable, but not actually modified. As such, due to Rust’s guarantees, I can abstract out the mutable/immutable variable distinction out of the bijls compiler.

However, there are some exceptions to this rule, and we cannot simply rely on Rust’s compile time checks passing. In the case of Rust types with interior mutability, an immutable value can have an interior, mutable value (allowed by types such as RefCell and Mutex). This means that there are cases where multiple references to the value can exist (as they are immutable), but as they can mutate an interior value, they break the rule of only allowing one mutable reference at a time. In those cases, since the ownership rules are checked in runtime, as opposed to compile time, the ownership and borrowing rules will need to be written in as part of the type representation.

For the purpose of this project, I consider this behaviour to be out of scope. The rest of the system at the current time does not need to consider
the ownership and type systems in such a manner. There is also a practical consideration here, as the project needed to be restricted to what could be reasonably implemented along with other features with the given time, resources and conditions. However, this is a potential area of future work arising from this project and topic.

3 Tool Implementation

The tool, named **rust2bils**, is constructed in Rust, using the **syn** Rust parsing library. Using the library, the compiler receives an AST (abstract syntax tree) of the code, which it proceeds to use in order to recursively build up the bils result. The compiler walks through the AST during the compilation progress along with a current state representation, which is used to recognise particular cases or adjustments that need to be made to account for various cases in bils syntax that don’t easily arise from equivalent Rust syntax, or for other features like automatic indentation.

The tool also contains additional features such as a "dual" mode, which takes in two Rust source files and compiles them into a single bils file, which is ready to be checked with the Hobbit tool. This is paired with a bils type resolver, the result of which is attached to a dual mode file in order to resolve potential cases where the Hobbit tool cannot automatically determine the full type. In addition, there is also a debugging feature which, instead of compiling to bils, outputs the AST of the input file. This was used to help development and examine particular Rust cases where the tool required additional work to correctly compile into bils.

As a full example of the compilation process, here is a simple function that mathematically swaps two integers, the compilation process of which I will go through in detail:

```rust
fn int_swap(mut a: i32, mut b: i32) -> (i32, i32) {
    a = a - b;
    b = a + b;
    a = b - a;
    (a, b)
}
```

Listing 17: Rust integer swap example

The first step of the compilation process is the system taking in the source file, parsing it with the syn library, and receiving an AST of the code. The AST contains the full contents of the file, the top level of which in this case is very minimal - a list of one item, the **int_swap** function. The AST then gets passed into the compilation process.

The first thing that the compiler does in the process at this time to reverse the list of file contents - this is done as a result of the considerations on code layout from the language mapping section. In order to simplify development, the compiler currently takes in files with the assumption that the first function that appears in the file is the "main" function, for which it is compiling a bils
expression. In addition, the compiler assumes that any function that is used by another function is be defined after it in the code.

While these rules are not necessary when writing Rust code, this spared us the effort of having to write additional logic for rearranging the code for bils compilation. Following from this, the compiler reverse the list to get the resulting nested definition of a lower function being in the context of a higher one. As such, in the case of multi-function files, the compiler successfully deal with the nesting of definitions.

```rust
fn increment(y: i32) -> i32 {
    y = plus_one(y);
    y
}
```

Listing 18: Rust multi-function ordering example

```bils
# plus_one
let rec plus_one params = (
    let num__prep = params in
    !num + 1
) in
# increment
fun params -> (
    let y__prep = params in
    y := plus_one( !y );
    !y
)
```

Listing 19: bils multi-function result

The compiler then proceeds to read over and process the function's parameter list. As stated in the mapping section, I model a function's parameters as a tuple, with additional boilerplate for ref definitions (Listing 3). The compiler collects the parameter names, attaches the _prep postfix for tuple unpacking, and then proceeds to define a new variable for each of the parameters. This would also be the location where things like the reference mapping would be accounted for, making sure that the syntax for extracting the getter/setter tuple is consistent and correct.

```rust
fn int_swap(mut a: i32, mut b: i32) -> (i32, i32) {
}
```

Listing 20: Rust integer swap function signature

```bils
# int_swap
fun params -> ( let ( a__prep, b__prep ) = params in
    ref a = a__prep in
    ref b = b__prep in
)
```

Listing 21: bils resulting function signature boilerplate

After this, the compiler moves onto the program body. It begins by performing a (mostly) generic processing of a code block, such that the process is similar for blocks in if/else statements and loops. However, due to the mapping, various cases and checks have to be looked at in order to ensure the compilation makes sense within bils syntax. These cases include taking different action at the end of a code block, to ensure that the function is returning the correct value, ensuring that if a particular getter/setter function needs to be used instead of the standard syntax in the case of arrays and
references (or other potential future types), or resolving between the differences in usage between `let` and `ref` definitions.

In this simpler case, the primary case is the latter, as locations need to be dereferenced with the `!` symbol, in addition to changing the value with `:=`, as opposed to `=` being used in other definitions and declarations. In addition, I have to suspend the semicolons marking each statement, to ensure that the last one (the new tuple of the two values) acts as a return value as opposed to a sequenced operation.

```
1 a := !a - !b;
2 b := !a + !b;
3 a := !b - !a;
4 ( !a, !b )
```

Listing 22: bils syntax for operating with `ref` variables

In other cases, this would also extend to generating the boilerplate for the reference (Listing 16) and arrays (Listing 14), shown in the mapping section. Extending this example, take a case where I compared the integer swap function with an erroneous version, with `b = a - b;` instead of `b = a + b;`. For this case, I would take the two examples, and use them with the dual mode of the compiler. The dual mode would first individually go over the above process with both files, resulting in the raw individual compilations into bils for each function. At this point, the dual mode will in addition run side functions to determine the type of the end expression(s).

The process reads over the signature and parameters of a function, and maps it into the resulting bils type. This process also needs to take into account the differences I have made in mapping concepts like arrays and references. For example, raw arrays have to be represented as full tuples, while references need to take into account that they are functions of particular types, and not just the original type with a marker, as it is in Rust. Taking the following example of a binary search function signature, I have to properly represent the getter/setter tuple that takes the place of the original's array reference:

```
fn binary_search ( array : &[ i32 ; 10], item : i32 ) -> bool {

Listing 23: Rust binary search function signature
```

This expression type is added to the marker separating the two bils expressions that constitute a full bils comparison pair. Given our integer swap example, this is the final result from the dual mode:

```
# int_swap
fun params -> ( let ( a__prep , b__prep ) = params in
  ref a = a__prep in
  ref b = b__prep in
  a := !a - !b;
  b := !a + !b;
  a := !b - !a;

Listing 24: bils expression type for the binary search function
```

This expression type is added to the marker separating the two bils expressions that constitute a full bils comparison pair.
After applying the file to the Hobbit tool, it correctly fails to find an equivalence between the two expressions, which shows it has detected the difference in the two functions.

4 Limitations

The compiler has a number of limitations, with most of them coming from potential features being cut out of the current state of the project because of extenuating circumstances meaning I was limited in the amount of work I could perform on the project. In addition to this, there are also some issues stemming from the Hobbit tool itself, and general, unavoidable limitations that could apply to a project such as this.

4.1 Rust Features

There are a number of Rust features that could reasonably be, and some of which were originally planned to be, implemented into the compiler:

**File Organisation:** As described in the implementation section, the tool currently needs the files and functions to be in a particular ordering. With additional work, this requirement could be relaxed, with the compiler simply reading over the code, constructing a dependency graph for the target function, and arranging the compiled result according to the dependency graph.

**Structs:** While I do mention mapping the struct layout in the mapping of compound types, the tool does not yet actually support this mapping, mainly because I have not implemented the associated struct methods:
```rust
let example = Example {a: 1, b: 2}; // struct instantiation
example.method(); // method call
```

Listing 26: Rust struct and method example

In Rust, methods are defined in an `impl` block, and non-static ones take a parameter of `self`, `&self` or `&mut self`, matching with the Rust ownership type system. As such, keeping the tuple syntax described before, it is reasonable for methods to simply be defined as additional functions in the nested context, with the necessary adjustments to ensure that the proper type/reference is passed when the method is called.

**Iterators and Ranges:** These features are commonly used in Rust, especially as part of loops, in order to complete the standard `for` loop construct.

```rust
let array = [1, 2, 3, 4, 5];
let length = 5;
for element in array.iter() {
    // iterator loop example
}
for index in 0..length {
    // range loop example
}
```

Listing 27: Rust iterator and range example

While their general implementation might require much more work for a fully correct translation, it would be reasonable to implement them being used as part of loops, as they would be used more than simply `while` loops with manual changing of variables. This could take the more primitive mapping of simply translating them into an equivalent operation in a `while` loop, or a more fully correct one with a complete `next` function providing the next element in the iterator.

**Closures:** Closures are effectively functions that can be saved, used, passed and returned as a variable in code, and which can also read other variables in the scope that they are defined in (subject to Rust’s ownership type system). While not as universal as other features mentioned here, they are often used with other systems that add functionality to these in-place functions.

```rust
let plus_one = |num| { num + 1};
let four = plus_one(3);
```

Listing 28: Rust closure example

Closures naturally map onto bils, as bils makes no distinction between a "static" function and one defined as a variable. As such, they can be reasonably easily added to the compiler, to extend its overall capabilities.
4.2 The Hobbit Tool & Other Considerations

There are also some limitations that arise from issues with the Hobbit tool. The main issue right now is problems with the syntactic sugar for arrays, especially with operations that require writing to or changing an array. As such, the compiler cannot be certain that it is correctly processing any examples that do so, such as sorting algorithms, until such a time as when the Hobbit tool can resolve these issues, or offer an effective workaround.

Finally, there are general, unavoidable limitations with a compiler such as this, arising from the nature of the problem. The tool cannot work with ”black boxes” or programs it does not have code for, as in those cases it has no information that it could potentially work on, and cannot anticipate any potential side effects that it cannot read for. A similar case can be for interactions that are unpredictable, or outside influences like the operating system - all of these cases are simply outside of the scope of the compiler and verifier.

The proper procedure for these issues can be things such as models of external systems, e.g. abstract representation of basic I/O with parameter functions that either give or consume information, with compatible representation of their operations. However, as a general rule, the main thing that can be done is to ensure that whenever the compiler and Hobbit tool are used, the user and any involved systems are aware of these unavoidable limitations, and to work within them.

5 Evaluation

For evaluating the tool, I have collected a set of reasonable test cases, which act as examples of the features of Rust that the compiler tool is compiling into bils. Outside of what was mentioned in the section on limitations, all of the features described in the mapping section have examples within the test cases, and are represented such that they can successfully be verified with equivalent results and functions using the Hobbit tool. These examples can be found in the code_examples folder as part of the compiler tool’s repository.

One of the most thorough test cases that I have compiled and are able to verify with the tool is an implementation of the binary search algorithm, which takes in a reference to a fixed length sorted array, and returns if the element is present in the list.

```rs
fn binary_search(array: &[i32; 10], item: i32) -> bool {
  let mut lo = 0;
  let mut hi = 10; // array length
  while (lo + 1) < hi {
    let mid = (lo + hi) / 2;
    let curr = array[mid];
    if curr <= item {
      lo = mid;
    } else {
```
Listing 29: Binary search implementation

This particular example is based on the bils-native implementations of binary search that were written by the Hobbit tool developers for testing the tool. It was tested within Rust and no apparent errors with the code were found. The resulting bils output for the above is as follows:

Listing 30: Binary search mapped to bils

With this, I have managed to detect inequivalency arising from various errors that I introduce to the base binary search implementation, by taking the implementation, introducing the error, and comparing against the original implementation by compiling the two versions into bils and running them with Hobbit. The system is able to detect when I introduce errors by:

- Changing the comparison operators for the loop or inner if to a different meaning (e.g. < as opposed to <=)
- Changing the various constants in the code to differing values (e.g. changing the initial hi value)
- Adding operations to the algorithm that modify the variables unnecessarily (e.g. lo = mid + 1;)

In addition to this, the example can be cross compared with the initial bils-native implementation that it was based on, given additional definitions to match with the function structure and the reference format. When this is done, the two examples are verified by the tool as equivalent, even though they differ by being iterative (the Rust implementation) and recursive (the bils-native
implementation), and the Rust-compiled version has the additional definitions of references involved. The two methods also successfully detect inequivalency when errors are introduced in either implementation by the methods mentioned above.

I have also written a bubble sort and an insertion sort implementation in the compatible subset of Rust. In this case, the sorting function takes in the entire array, as opposed to the binary search example, which only takes in a reference to the array.

```rust
fn bubble_sort(mut array: [i32; 10]) -> [i32; 10] {
    let mut swap = true;
    // keep repeating until a swap does not occur.
    while swap {
        swap = false;
        let mut index = 0;
        while index < 10 - 1 {
            if array[index] > array[index + 1] {
                swap = true;
                let swaptemp = array[index + 1];
                array[index + 1] = array[index];
                array[index] = swaptemp;
            }
            index += 1;
        }
    }
    array
}
```

Listing 31: Rust bubble sort implementation

```haskell
# bubble_sort
fun params -> ( let array__prep = params in
    ref array = array__prep in
    let array__get i = if (i < 0 || i >= 10) then _bot_ else
        if i = 0 then ! array[0/10] else if i = 1 then ! array[1/10] else
            if i = 2 then ! array[2/10] else if i = 3 then ! array[3/10] else
                if i = 4 then ! array[4/10] else if i = 5 then ! array[5/10] else
                    if i = 6 then ! array[6/10] else if i = 7 then ! array[7/10] else
                        if i = 8 then ! array[8/10] else if i = 9 then ! array[9/10] else
                            _bot_ in
        let array__set i = fun m -> if (i < 0 || i >= 10) then _bot_ else
            if i = 0 then array := ! array[0/10]:= m] else
                if i = 1 then array := ! array[1/10]:= m] else
                    if i = 2 then array := ! array[2/10]:= m] else
                        if i = 3 then array := ! array[3/10]:= m] else
                            if i = 4 then array := ! array[4/10]:= m] else
                                if i = 5 then array := ! array[5/10]:= m] else
                                    if i = 6 then array := ! array[6/10]:= m] else
                                        if i = 7 then array := ! array[7/10]:= m] else
                                            if i = 8 then array := ! array[8/10]:= m] else
                                                if i = 9 then array := ! array[9/10]:= m] else
                                                    _bot_ in
                                        ref swap = true in
                                            let rec loop () = ( if (!swap) then ( swap := false;
                                                ref index = 0 in
```
Due to not using a reference and instead taking in the whole array, the tool rebuilds the syntactic sugar for operating on arrays. However, Hobbit is currently not working correctly with the particular syntactic sugar for arrays, which is preventing it from checking sorting algorithms, as noted in the section on limitations. The performance of the tool is relatively quick, when compared to actual program compilation into binary, or proceeding to use the compiled bils output with the Hobbit tool. All of the compilable snippets are consistently compiled on a computer running an Ubuntu 20.04 virtual machine, given 8GB of RAM, with an Intel Core i7 2.30GHz CPU, in 70-85ms, which probably indicates that the examples used are too small to impact the program’s running time. As an aside to check this further, I timed the command with a file containing successive powers of two repetitions of the binary search example on the same machine. This showed that the tool’s running time is linear to input size.

<table>
<thead>
<tr>
<th>Repetitions:</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s):</td>
<td>0.08</td>
<td>0.09</td>
<td>0.10</td>
<td>0.12</td>
<td>0.16</td>
<td>0.25</td>
<td>0.42</td>
<td>0.77</td>
<td>1.45</td>
</tr>
</tbody>
</table>

6 Future Work

The first and most present area of potential future work on this project and general area is extending the implemented Rust feature subset. For the most part, this would be the potential work that could have been done for this project, but had to be scoped out due to extenuating circumstances, as described in the section of limitations. This can include, but is not limited to the missing features noted in the section, along with more advanced Rust features, such as Vectors (dynamically sized arrays) and traits (akin to interfaces in other languages). If the compiler’s supported subset of Rust is expanded to a sufficient degree, it can likely be begun to be applied to more practical projects, at least in a limited capacity. As such, on an expanded compiler, work can be done alongside the Hobbit tool to attempt to integrate the entire checking process into a set of practical tools. For example, the comparison of certain functions, if they were changed between versions, could be integrated into a general testing environment.
There is also the potential for cross-compilation of other languages into bils. If similar compilers such as the one started in this project appear, they can be used simultaneously in order to run comparisons between languages, instead of just within Rust (or any other particular language). This can open the possibility of comparing code between languages, and the usage of the Hobbit tool in cases like attempting to port systems to a different language, either for modernisation, desired features, or simply easier support and development.

7 Conclusion

In this project, I have managed to map a subset of the Rust programming language into bils, the ML-like language used by the Hobbit (Higher Order Bounded BIsimulation Tool) tool, and develop a tool for compiling the target subset according to the mapping. In addition, using both tools, I have managed to prove equivalence and inequivalence between various Rust snippets within the target subset, checking and verifying that the compiler tool has managed to achieve its primary goal. I have also planned out potential avenues on which the tool can be improved upon, and even potentially approach being used in practice.

I consider the project to be successful, even if the resulting tool ended up somewhat limited. I have managed to compile using implementations of some standard algorithms, the most successful of which was the implementation of binary search. The binary search example being fully compiled, and able to detect the inequality between it and various errors I could induce on an incorrect version of the algorithm are a successful case of being able to use the compiler tool for the intended purpose of equivalence checking. Issues with the Hobbit tool prevented me from checking if the sorting algorithm compilation examples were correct, however I have managed to at least implement enough support such that the compilation for these types of algorithms can occur, and potentially checked once the issues with the Hobbit tool get resolved.

The fact that the compiler and tool could reach a state where it can work with these examples, leads me to believe that it could work on more practical examples. However, a lot of work needs to be done before that’s a real possibility. First, there is implementing the missing features to ensure as high compatibility with practical Rust projects. Second is making the compiler compatible with the standard code structure of Rust code projects. There is also the matter of having some model for abstractions such as I/O, though those can also be partially left to the user of the compiler once the abstractions are necessary.

However, if the compiler and the Hobbit tool were developed further, they could see potential use in assisting making patches and changes to systems, helping to guarantee correctness between patches. A functional system could potentially be used in practice, especially with Rust being increasingly used in systems programming. As such, I believe it likely that this type of project could see practical use in those scenarios. However, this may be limited depending on cases like the upper limits on possible compatibility with the code, but for this
to be determined, further projects into this may be required. In addition, more
work on the Hobbit tool, especially in the case of helping to interpret results in
the case of a detected inequivalence, would help the project as its users could
easier see what the detected point of inequivalence was.

My own experience with the project was mixed between working in an
interesting area of research that I have not seen much of in comparison to
other topics, and having to do so during a very inconvenient time and
situation.

Getting to work on a more theoretical, and relatively obscure topic, was
refreshing, as formal verification is not a topic that I often get to interact with.
During the project I gained a much higher understanding of what is involved
with the field, how it is used in critical circumstances, which are not touched by
most other fields of work in computer science. I personally am more interested
with the lower level and theoretical side of computing, and as such I feel that
this project has helped me gain a much more developed basis that I could use
for future work or endeavours.

However, this project was done in what was most likely one of the worst
set of circumstances I had to ever so significant work under. Between the
COVID-19 pandemic, a data loss occurring, and various personal issues
appearing throughout the time of the project, I was severely hindered in what
I could accomplish. Going into the last year and this project, I had assumed
that from my interest in the field and my solid academic record, I would easily
be able to deal with the circumstances and get the project done to my
original, more considerable goals. But as the academic year continued, I had
to ultimately accept the toll that the situation took on me, and reduce the size
of the project to what I could more reasonably accomplish, as I could not
simply power through all these difficulties just by my academic ability alone.

At this point I am satisfied with the fact that I at managed to take the
project to the state it is in. Though limited, I reached the goal set out and,
while it cannot really be represented in the work alone, I managed to do so
through very unproductive circumstances. I hope that the work that I have
accomplished can serve as a useful point for continued research into this area.

References


Again: From Bounded Checking to Verification of Program Equivalence via
