Extraction of Rust code from the Why3 verification platform

Nils Fitinghoff

Computer Science and Engineering, master's level
2019

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Abstract

It is hard to ensure correctness as software grows more complex. There are many ways to tackle this problem. Improved documentation can prevent misunderstandings what an interface does. Well built abstractions can prevent some kinds of misuse.

Tests can find errors, but unless they are exhaustive, they can never guarantee the absence of errors.

The use of formal methods can improve the reliability of software. This work uses the Why3 program verification platform to produce Rust code. Possible semantic-preserving mappings from WhyML to Rust are evaluated, and a subset of the mappings are automated using Why3’s extraction framework.
Acknowledgement

I would like to thank my supervisor Professor Per Lindgren for his patience and encouragement.
## Contents

1 Introduction .................................................. 4  
   1.1 Rust .................................................... 5  
      1.1.1 Ownership and borrowing .......................... 6  
   1.2 Problem definition and delimitations ..................... 6  
   1.3 Method .................................................. 6  
   1.4 Thesis structure ........................................ 6  

2 Mappings from WhyML to Rust ................................. 7  
   2.1 Appending two lists ..................................... 7  
   2.2 Identifiers .............................................. 9  
   2.3 Pattern matching ....................................... 9  
   2.4 Types .................................................. 11  
   2.5 Mutability and record field assignment ................. 13  
   2.6 Blocks .................................................. 14  
   2.7 Conditional expressions ............................... 14  
   2.8 Loops .................................................. 14  
   2.9 Functions .............................................. 14  
      2.9.1 Abstract functions ................................. 15  
      2.9.2 Closures ........................................... 15  
   2.10 Exceptions ............................................. 15  
   2.11 Submodules ............................................ 17  

3 Automated translation ......................................... 18  
   3.1 Extraction architecture .................................. 18  
   3.2 Pattern matching ....................................... 20  
      3.2.1 Problems with let ................................. 20  
   3.3 Types .................................................. 20  
   3.4 Functions .............................................. 21  
   3.5 Mutability and record field assignment .................. 21  
   3.6 Borrowing and references ................................ 21  
   3.7 Exceptions ............................................. 22  
   3.8 Formatting .............................................. 22  
   3.9 Derive .................................................. 22  

4 Evaluation ................................................... 24  
   4.1 Overview ............................................... 24  
   4.2 Surprise exceptions ..................................... 24  
   4.3 Function application bug ................................ 24
Chapter 1

Introduction

More and more things around us are software-controlled – from IoT fridges to pacemakers to our schedules. We rely on these things to work, and the consequences when things go wrong vary from the inconvenient to the lethal. The problem is that creating reliable software systems is difficult. There are many ways to try to tackle this problem, with methods that offer stronger guarantees often being more difficult and time-consuming to apply.

- **Extensive testing** can discover situations where the software does not behave as expected. It cannot, however, be certain to find all such situations, unless every possible input is considered.

- **Fuzzing** or **fuzz testing** generates input to the program automatically in order to find interesting behavior. This means that it is harder to know if the program output is correct, crashing is often the only way to find failure. Assertions can be used to express some of the expectations on the program.

- With **symbolic execution**, all paths through the program can be spanned to generate tests, or to prove that some assertions in the program are always true. This allows more properties of the program to be verified, as it can be exhaustive up to the program paths followed.

Moving further towards stronger guarantees requires stronger specifications. Instead of specifying some input/output pairs to be correct, as is done in testing, it is desirable to describe all the properties the program needs to have, for all inputs. Specifications of this kind often cannot be expressed well in programming languages. Equipped with a specification of the program, it is possible to verify that the program follows the specification using different techniques. This work uses the Why3 platform for **deductive program verification**. Its associated language WhyML \[9,8\] can express both specifications and their implementations. Verification conditions are generated automatically based on Hoare logic \[11\], and proofs are facilitated by interfacing with a wide range of automatic SMT solvers and interactive theorem provers.

Deductive program verification falls in the broader category of formal verification. The program (annotated with specification) is used to generate proof obligations which, if proven, guarantee that the program complies with the
specification. The proof process in Why3 allows the user to transform the proof obligation (e.g. to perform induction or to split a conjunction into multiple goals to prove). The strength of Why3, however, is its integration with external provers. In particular, proof obligations can be translated into input for SMT solvers to automatically prove the goal. Difficult proof obligations can be transformed into input for interactive theorem provers, e.g. Coq [4].

There are two different workflows that are used to produce verified programs using Why3:

- Implementations written in some other language can be modeled in Why3 to facilitate proofs. This can be done either by hand such as in [7] or automatically (using annotations in the source code to specify the properties to be proven) using a framework such as SPARK [3] or the Jessie plugin for Frama-C [16, 1]. This is not unique to the Why3 platform, for example the same approach is taken in [21] to verify Rust programs using LEAN [2].

- Implementations written in WhyML with proven properties can be translated to an equivalent program in some other language. This is called extraction, described in [17] with OCaml as target. Extraction is also used in other verification frameworks such as Coq [10].

1.1 Rust

Rust [19] is a programming language that emphasises performance and reliability. Its memory model based around ownership and its type system together prevent many common bugs.

Development is lead by Mozilla, where the language is used to replace components of the Firefox web browser. This necessitates a good foreign function interface, to avoid rewriting the whole program at once (as it would carry too great of a development cost).

Similarly, formal verification of programs can be both difficult and time consuming. For this reason, it may only be viable to make the extra effort of verifying some select components of a software system. Then, it is desirable to express the rest of the system in a language that prevents unproved code from interfering with the proved components (for instance, through memory unsafety). Rust makes a good candidate for the lower-assurance components of the system. Then, taking the second approach to producing verified programs using Why3, Rust is an interesting language to target.

This is not the first effort towards formally verified programs in Rust. The RustBelt project [6, 12] investigates and attempts to verify Rust’s safety claims, especially focusing on “verifying safe encapsulation of “unsafe” code”.

Symbolic execution in KLEE has been used to verify properties of Rust programs, including WCET analyses [14] and assertion-based contract verification [15].

Automated translation to LEAN [21] has also been used to verify properties of Rust.
1.1.1 Ownership and borrowing

Ownership [20] is a central concept in Rust. Briefly, each value is owned by a single variable and values are dropped when their owner is no longer in scope. Values can be moved by a variable assignment or a function call; this transfers ownership. Types implementing the `Copy` trait are copied instead of moving ownership.

Using & references to borrow the value, ownership stays with the original variable, but allows other users access to the value. Rust places restrictions on the references that are allowed to exist at the same time in order to make its safety guarantees. A single mutable reference or no mutable references and any number of immutable references may exist at the same time. These restrictions on aliasing are checked statically.

1.2 Problem definition and delimitations

This work focuses on techniques for producing a Rust implementation that follows a WhyML specification to improve the reliability of the software, that is, make sure it has the desired behavior when used in the intended way. This is done by studying and evaluating possible semantic-preserving mappings from WhyML to Rust.

In particular, we need to investigate, evaluate and implement solutions for semantic-preserving mappings regarding

- Memory model,
- Type system,
- Module system and
- Syntax.

Robustness, limiting the severity of failures when faced with unexpected input, is not part of this work, as it is not about ways to write good specifications. Methods for creating the proofs that ensure that the implementation follows the specification are not part of the work either.

1.3 Method

A background study builds an understanding of how the Rust and WhyML work, and how extraction is done in Why3, then we study and evaluate possible semantic-preserving mappings from WhyML to Rust and finally implement an automated mapping for a subset of WhyML.

1.4 Thesis structure

In chapter 2 differences in semantics and possible mappings from program constructs in WhyML to Rust are discussed. Chapter 3 details implementation choices in the automated translation, and chapter 5 ties the results together.
Chapter 2

Mappings from WhyML to Rust

Here possible mappings will be discussed. First a small example illustrates how a WhyML function can be translated to Rust, and some of the differences between the languages. Afterwards possible mappings for specific language constructs are discussed.

2.1 Appending two lists

The Why3 standard library provides a function that appends two lists, as seen in listing 2.1. Here, some details are removed. A more complete listing is provided in appendix A.

```whyml
let rec function (++) (l1 l2: list 'a) : list 'a = 
  match l1 with
  | Nil -> l2
  | Cons x1 r1 -> Cons x1 (r1 ++ l2)
end
```

Listing 2.1: Standard library list and append

The use of both `let` and `function` mean that `++` is introduced both as an implementation and as a logic specification. A number of additional properties to be proved are added using lemmas, such as associativity:

```whyml
lemma Append_assoc:
  forall l1 @induction] l2 l3: list 'a.
  l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3
```

Listing 2.2: Associativity of `++`

These properties are useful in Why3 when proving properties of code that appends lists, but since we translate only the implementation, not the specification, the lemmas are ignored.
First, consider the **list** type definition. It has a generic type parameter `‘a` and is recursive in the **Cons** case. In Rust, this would correspond to an **enum** with a type parameter `A`.

```rust
enum List<A> {
    Nil,
    Cons(A, List<A>),
}
```

**Listing 2.3: Faulty Rust definition of list**

This definition fails to compile because the Rust requires that each type must lie in one memory allocation. This means that Rust has to find an upper bound on the size of the type. The recursive definition means that the type can be of infinite size. In order to make the type representable, indirection needs to be inserted so that the recursion is broken up by a pointer to a different allocation. This can be done in different ways. WhyML does not restrict the type to reside in a single memory allocation, and as a result, can add indirection behind the scenes.

Using `&List` references would require a lifetime annotation to indicate that the contained reference must be valid for at least as long as the containing `List` exists. In this case, `++` only reads the old list to construct a new one, but sometimes a mutable reference might be needed.

Another alternative is to use a `Box<List<A>>`. This is a heap allocation containing a `List<A>`. With this, the definition becomes

```rust
enum List<A> {
    Nil,
    Cons(A, Box<List<A>>),
}
```

**Listing 2.4: Rust definition of list using Box**

With this, we can move on to the `++` append function. Infix functions in Rust are limited to overloading operators for custom types, e.g. `std::ops::Add`. Rust has a rule that prevents implementing traits for types if neither the trait or type is defined locally, so if the arguments of the WhyML infix function are mapped to types in Rust’s standard library, the `ops` overload will not work.

Instead, we can simply use prefix functions. Since the name `++` is not valid in Rust, we can use `plusplus`, or, since we know what the function appends two lists, simply `append`.

The generic element type (`‘a` in WhyML) must be declared in the function signature in Rust, capitalizing to follow style recommendations. The `‘` is not present in the Rust version because it is WhyML syntax and not part of the identifier.

For the function parameters, there is a choice between taking ownership of the two lists and borrowing them. In this case, borrowing is semantically closer to the WhyML code, since it allows the caller to continue using the lists after creating the new list. The Rust `Clone` trait provides a way to get a copy of the value, requiring an explicit call to the `clone()` method. Implicit copying comes with the `Copy` trait, which is limited to bitwise copying of the value. This means that some types cannot implement `Copy` (e.g. types that contain mutable references to heap allocations).
fn append<A: Clone>(l1: &List<A>, l2: &List<A>) -> List<A> {
    match l1 {
        List::Nil => l2.clone(),
        List::Cons(x1, box r1) =>
            List::Cons(x1.clone(), Box::new(append(r1, l2))),
    }
}

Listing 2.5: append using borrowed values

Borrowing will create some complications with the return value. In the Nil case of the match, a copy of l2 must be returned. This means that List should derive Clone, requiring that the contained item also is Clone. This in turn forces append to bound its type parameter with Clone. Listing 2.5 borrows its input lists, however owned values are easier to deal with, giving us listing 2.6.

fn append<A>(l1: List<A>, l2: List<A>) -> List<A> {
    match l1 {
        List::Nil => l2,
        List::Cons(x1, box r1) =>
            List::Cons(x1, Box::new(append(r1, l2))),
    }
}

Listing 2.6: append using owned values

In this simple case, the match mostly differs in syntax. Differences are discussed in more detail in section 2.3. Because the List uses a box, the match has to get the boxed value. The box syntax used here is unstable and hidden behind a feature gate `#![feature(box_patterns)]` that is only accessible on the nightly version of the compiler. It is possible to use a nested match to get the boxed value without using the feature gate.

2.2 Identifiers

The characters allowed in identifiers such as variable and function names differ between WhyML and Rust. WhyML allows the ASCII apostrophe `'` as part of identifiers while Rust does not. The languages reserve different identifiers as language keywords, too. For example, `trait` is a valid identifier in WhyML, but not in Rust.

In both these cases, the identifiers must be renamed when translating to Rust. Some care needs to be taken to ensure that the renamed identifiers don’t collide with other identifiers in the program.

2.3 Pattern matching

The syntax for pattern matching differs between the languages. In WhyML, pattern matching over `e` starts with `match e with` and ends with `end`, while Rust pattern matching doesn’t use `with` or `end` and encloses the patterns with `{ and }` instead.
In WhyML, match cases are listed as | pattern -> expression, while Rust uses pattern => { expression}.

WhyML has a variety of patterns that can be matched. Shown in listing 2.7, most of them have a Rust syntax equivalent.

<table>
<thead>
<tr>
<th>1</th>
<th>T0 (_, _) (* constructor *)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{a=a; b=b} (* record constructor (T1) *)</td>
</tr>
<tr>
<td>3</td>
<td>T2</td>
</tr>
<tr>
<td>4</td>
<td>(a, b) (* tuple *)</td>
</tr>
<tr>
<td>5</td>
<td>(a, b) as c (* c = (a, b) *)</td>
</tr>
<tr>
<td>6</td>
<td>a (* introduce variable *)</td>
</tr>
<tr>
<td>7</td>
<td>_ (* wildcard *)</td>
</tr>
</tbody>
</table>

Listing 2.7: Examples of WhyML patterns

In T1 | T2, the | means that this case should be applied if either case matches. The syntax is the same in Rust. Tuple patterns can omit the parentheses in WhyML but not in Rust. Introduction of variables to contain the components of the match is only different in the requirements on valid identifiers. The _ wildcard is the same in Rust in that it matches anything without binding it to a name.

The pattern for the constructor T0 is the same in Rust, but enum variants are in the type’s namespace, resulting in T::T0. If the variants are imported immediately after declaring the enum, this is not an issue.

The (record/struct) application of T1 has different syntax. Rust uses : and , in place of WhyML’s = and ;. A more interesting difference is that Rust requires the name of the type before the curly braces.

| 1 | T0(_, _) |
| 2 | T1 {a:a, b:b} |

Listing 2.8: Constructor application pattern in Rust

A pattern that is likely to cause trouble in translation is as because of its ability to create aliases. In the translation in listing 2.9, the compiler does not allow pattern bindings after @, because then c could be used to access the same data that a owns, which would break the ownership model.

| 1 | c @ (a, b) |

Listing 2.9: as pattern in Rust

It seems best to avoid this kind of pattern in the WhyML program to make translation easier. As long as no names are bound after the @, the translation works.

The special considerations with boxed types have already been discussed in section 2.1, but it is worth noting that similar problems can show up if the translation requires types to be wrapped. Box is just one example of this.

let variable bindings are a special case of pattern matching in both languages. WhyML uses let pattern = expr in expr, with bindings introduced by the pattern valid only in that second expr. In Rust, the binding is kept to the end of the containing scope, using the syntax let pattern = expression;.

For example, take this match over the option type from the standard library.
Listing 2.10: Non-exhaustive let pattern matching over the option type

Listing 2.10 binds the name v to the value 5. Since there are no alternative branches to take here, it is important that the pattern is exhaustive (it always matches). WhyML allows non-exhaustive patterns in program expressions, but generates a proof obligation to ensure that the values that are not covered cannot occur. Rust only allows non-exhaustive matches if it is able to prove that the values that are not covered cannot occur. This means that a direct translation of a (proved safe) non-exhaustive pattern can fail because the Rust compiler fails to find a proof.

It is possible to work around this by rewriting the offending non-exhaustive match with match, placing the rest of the code where the names need to be in scope in the match arm. match must also be exhaustive, so catch-all arm is still needed. Marking it unreachable allows arm to be optimized out by the compiler.

Listing 2.11: Working around the non-exhaustive pattern in listing 2.10

This translation is very cumbersome to read, but for some non-exhaustive cases the more readable let translation simply will not work, and it allows easier control over where the scope for the introduced v ends. The _ arm is guaranteed to be unreachable by the proof obligation generated by Why3, thus we can safely mark it unreachable in the Rust code.

2.4 Types

According to Rust style guidelines, types should be given CamelCase names. In WhyML, only some identifiers start with a capital letter, such as modules and constructors. Since Rust is less restrictive in the allowed identifiers, this poses no problem for translation.

Algebraic types become enumerated datatypes. The differences are syntactic.

Listing 2.12: Algebraic type (WhyML)
enum T<A> {
    A,
    B(A, A),
}

Listing 2.13: Enum definition (Rust)

Records become structs. The differences are syntactic.

type t 'a = {
    a: 'a;
    b: bool;
}

Listing 2.14: Record definition (WhyML)

struct T<A> {
    a: A,
    b: bool,
}

Listing 2.15: Struct definition (Rust)

Type aliases only differ by a semicolon: the WhyML alias type t2 = t1 is
expressed type T2 = T1; in Rust.

When functions appear in types, the Rust translation has a few different
options:

1. Use one of the closure traits: Fn, FnMut or FnOnce. They require increas-
ingly exclusive access to their environment. Fn borrows the environment,
   FnMut borrows mutably (so it requires exclusive access) and FnOnce takes
   ownership of the environment (and the closure cannot be called again).

2. Use a function pointer, fn(...) -> .... This prevents the use of closures
   entirely. Function pointers implement the three closure traits easily since
   they have no environment to capture.

   Given that WhyML functions can capture the environment, function pointers
   are not a good match. The need to share the closure should drive the decision
   of which closure trait is used.

   The problem of recursive types was already mentioned in section 2.1. It
   is also possible to have mutually recursive types, and then there can be many
   different places where the indirection can be added.

type a = B b
with b = C c
with c = A a | Nil

Listing 2.16: Mutually recursive types

Figure 2.1 views the types of listing 2.16 as a directed graph of type depen-
dencies. a can be constructed using a b, so there is an edge from a to b. Because
the graph has a cycle, the type has in Rust will have infinite size. If indirection
on the b to c edge is added, for example making b a boxed c, this corresponds
to removing the b-c edge in the graph. The modified graph is acyclic, so Rust
will be able to give it a finite size.
One could add indirection everywhere. It would be sure to remove any cycles, but each indirection adds some cost.

Another strategy is to add only as much indirection as is needed to remove all cycles. This requires finding the minimum feedback arc set of the graph, where the decision version of the problem is NP-complete [13].

A strategy in between the two could be a traversal of the graph that adds indirection on any edge that would complete a cycle.

In general, the best choice of which edges to remove by adding indirection is not obvious. Minimizing the number of indirections seems like a good goal, but requires knowledge of how the type will be used. In this example, adding the indirection in the reference from \textit{c} to \textit{a} requires one fewer indirection than the other two alternatives.

### 2.5 Mutability and record field assignment

Mutable record fields can be assigned new values using the <- operator, e.g. \texttt{t.t\_a <- a}. Rust has similar syntax for assigning a new value to a struct field, but there are differences in how mutability works. It can also be used to assign multiple struct fields at once using tuple notation, \texttt{t.t\_a, t.t\_b <- a, b}. In Rust, this would need to be written as two separate assignments, \texttt{t.t\_a = a;} and \texttt{t.t\_b = b;}.

The only way to get mutable data in WhyML is to mark a record field with the \texttt{mutable} keyword. WhyML mutability is granular on the record field level.

In Rust, the mutability syntax and support for static checking is on the level of one variable. Variables are declared mutable with the \texttt{mut} keyword.

Rust also has a concept of interior mutability, where select struct fields
can be mutable without making the whole struct mutable using the `std::cell` module. This circumvents the normal alias analysis (see section 1.1.1) done by the compiler, so run-time checks are inserted to prevent mutable aliasing. If the run-time checks fail, the program panics. There is also an unchecked `UnsafeCell` that can be used if there is some other way to know that the cell will not be used to break the aliasing rules.

While interior mutability using cells are the best match for WhyML’s mutable fields, if the types that use interior mutability are exposed to a user of the translated code, there is a risk of misuse. The result would be either panics (if run time checked `Cell` or `RefCell` are used) or undefined behavior (with `UnsafeCell`). Alias analyses on the Why3 side can not cover how the types are handled after translation.

### 2.6 Blocks

Expression blocks in WhyML are delimited by the keywords `begin` and `end`. The Rust construct that immediately comes to mind is `{ }` blocks, but Rust memory management creates a difference. If the Rust block due to other translation decisions returns a reference to data that is dropped at the end of the block, compilation fails. This may not be a problem in practice if references are used sparingly.

### 2.7 Conditional expressions

In both WhyML and Rust, the conditional `if c then e1 else e2` constructs are expressions. The difference is syntactic as the Rust translation only needs to add curly braces, resulting in `if c then {e1} else {e2}`.

### 2.8 Loops

While loops take the form of `while c do e done`, which is only syntactically different from the Rust while loops: `while c {e}`.

WhyML for loops are restricted to counting up or down for a fixed number of steps. For Rust’s native integer types this can be done using a for loop over an iterator created using the range operator `a .. b`. The trait to overload the range operator is not stabilized, so library types like `num::BigInt` implement their own methods to get range iterators.

### 2.9 Functions

WhyML function application is curried, so that the two definitions in listing 2.17 both have the same syntax `f 1 2` and `g 1 2` when called:

```whyml
let f a b : bool = a < b

let g a : bool = f a
```

Listing 2.17: Two functions that are called the same way in WhyML
In Rust, this is not the case, as each function takes a tuple containing its elements. To apply \( g \) in listing 2.18, one has to write \( g(1)(2) \), whereas \( f \) is applied as the more familiar \( f(1, 2) \).

```rust
1
2 fn f(a: i32, b: i32) -> bool {
3     a < b
4 }
5
6 fn g(a: i32) -> impl Fn(i32) -> bool {
7     move |b| f(a, b)
8 }
```

Listing 2.18: Rust translation of listing 2.17

### 2.9.1 Abstract functions

The WhyML keyword `val` can introduce a function with no body. It is often used by modules to take parameters. The closest thing in Rust would be external functions, which have signatures but no definitions and are used foreign function interfaces (FFI). This does not appear to be a very useful translation to make.

### 2.9.2 Closures

WhyML anonymous functions using the keyword `fun` can capture their environment, so the obvious choice is to translate them to Rust closures. Conveniently, closure signatures are allowed to use type inference.

```rust
1 fun a -> a
```

Listing 2.19: Identity function using `fun`

```rust
1 |a| a
```

Listing 2.20: Rust translation of the identity function

Functions defined in expression position (that is, using `let .. in` rather than the declaration `let ..`) can also capture their environment, so the best matching Rust construct is again a closure.

### 2.10 Exceptions

Exceptions do not exist in Rust. One possibility is to map exceptions to `panic!`, resulting in a stack unwind that can be caught by `std::panic::catch_unwind` (provided that the panic unwind behavior is not replaced). Panic behavior is not standardized on all platforms. In particular, `#[no_std]` applications such as embedded systems may use a simpler panic handler to avoid the overhead of supporting unwinding. To quote the documentation of `std::panic::catch_unwind`, “it is not recommended to use this function for a general try/catch mechanism.”

Instead, failure is typically handled in the type system using the `Result` type. Here translation becomes more difficult as the types of not only the function that may raise exceptions needs to change type signature, but also any code that calls that function.
Some things make the situation a little better. WhyML functions are required to declare any exceptions they can raise in their signature using the `raises` keyword. Exceptions are similar to types in that they are declared and can contain data. A Rust translation can take advantage of this to inform the decision of what error type to use in the `Result`.

```whyml
definition FooException is int
end

let f (a: bool) : int =
  raises { FooException }
  = if a then raise (FooException 3)
  else 9

let g (a: bool) : int =
  try f(a) with
    | FooException n -> n
  end
```

Listing 2.21: Exception declaration and use in WhyML

In the Rust translation in listing 2.22, the declaration of the exception has been replaced by an enum listing all possible exceptions. This allows the `Err` variant of `Result`s to hold any exception, and allows pattern matching to be used in place of WhyML’s `try with` syntax.

The function signature is changed to return a `Result`. The whole function body is wrapped in the `Ok` variant to wrap the return type in the non-exception case.

The `raise` expression is replaced by constructing the `Err` variant of the right exception expression and then a `?` operator that returns from the function if the operand is an `Err`. This will always be the case, and the reason `?` is used instead of `return` is to allow a local WhyML `try with` to translate to a `catch` block.

```rust
eenum Exceptions {
  FooException(i32),
}

def f(a: bool) -> Result<i32, Exceptions> {
  Ok(if a {
    Err(Exceptions::FooException(3))?
  } else {
    9
  })
}
```

Listing 2.22: Rust translation of listing 2.21

Another solution would be to refactor the WhyML code, replacing exceptions with a `Result` type. It allows less existing WhyML code to be translated without changes.
2.11 Submodules

The `scope` keyword creates a named submodule. In Rust, `mod` is used to the same effect.

```rust
scope S {
  let f a = a + 1
}
let g = S.f
```

Listing 2.23: Submodule use in WhyML

Rust is written very similarly. Note that the access to the new namespace uses `::` in Rust and `.` in WhyML.

```rust
mod S {
  fn f(a: u32) -> u32 { a + 1 }
}
fn g(a: u32) -> { S::f(a) }
```

Listing 2.24: Submodule use in Rust

Another possibility is to avoid creating a module for `scope` and disambiguate by renaming instead. It reduces readability by removing structure, so it is not very desirable.
Chapter 3

Automated translation

3.1 Extraction architecture

Extraction in Why3 is quite extensible. The WhyML source is converted into an internal representation. It is then simplified into a list of MLTree declarations, removing much of the information that is only present for analysis.

Extraction then continues based on a provided ‘driver’ file. The driver file specifies a printer to use, which must be registered to the printer driver. Printers are OCaml programs that convert MLTree declarations into strings. The driver can also contain mappings from WhyML symbols to arbitrary strings. This is often used when the target language has special syntax for something that is a function in WhyML. An example from the ocaml64 driver is the mapping of BuiltIn in listing 3.1

```
theory BuiltIn
  syntax type int "Z.t"
  syntax predicate (=) "%1 = %2"
end
```

Listing 3.1: Driver entry for the special WhyML module BuiltIn

The built-in WhyML integer type is mapped to the integer type from the zarith library and the WhyML integer equality predicate is mapped to OCaml structural equality. The printer has access to these mappings and is responsible for applying them. Positional arguments %1 and %2 are substituted by the printer.

The registered printing function is invoked on each MLTree declaration to produce source code in the output language. The path from WhyML source code to source code in the target language is shown in figure 3.1.

1 The solver input for proofs are generated in a similar way. The WhyML internal representation is converted into a list of proof tasks, to which any requested transformations are applied. Finally, a printer-driver similar to that for extraction generates the input to the solver.
Figure 3.1: The representations a program has during extraction.
3.2 Pattern matching

Pattern matching is extracted as described in section 2.3. The problem with the potential compile errors from the translation of \texttt{as} patterns to \texttt{\_} is left unresolved.

The cases where \texttt{let} is a non-exhaustive pattern match becomes the same pattern matching structure in \texttt{MLTree} representation, so no special case is needed to deal with them. The run time checked version \texttt{unreachable!} is used rather than the unsafe optimization hint \texttt{std::hint::unreachable_unchecked!} for the absurd case that is required to make the match exhaustive. This is perhaps overly cautious.

3.2.1 Problems with \texttt{let}

Some WhyML expressions must be translated to an expression block in Rust, such as in listing 3.2

\begin{verbatim}
1 let a = let b = 1 in b + b in f a
\end{verbatim}

Listing 3.2: \texttt{let} binding that corresponds to multiple Rust expressions.

In Rust, it becomes listing 3.3

\begin{verbatim}
1 let a = {
2    let b = 1;
3    b + b
4    };
5    f(a)
\end{verbatim}

Listing 3.3: Rust translation of listing 3.2

The extraction strategy for all \texttt{let .. in} is to produce expression blocks. This can become a problem because values created inside the scope are dropped at the end (unless they are moved when the block returns). This prevents expressions from returning references in this context as the compiler rejects references that outlive their referenced data.

The strategy could be refined to avoid inserting the blocks for simple \texttt{let} uses. This would remove the problem in the common case. Because of the decisions surrounding \texttt{ref}, only driver substitutions can currently produce code that returns a reference. Consequently, there has been little need to add any special cases.

3.3 Types

As discussed in section 2.4, recursive types require that some indirection is added. The chosen indirection is \texttt{Box} because it does not require lifetime annotations, making definition and use of the types easier than if references were used.

Without knowing in detail how the type will be used, it is not clear what choice is best. Thus, a relatively simple strategy was chosen. When (mutually) recursive types are encountered, the declarations are explored depth-first, boxing only when it would otherwise create a cycle.
3.4 Functions

Consider the WhyML code:

```whyml
use int.Int

let f a b = a + b
let g a = f a
let v () = g 1 2
```

Listing 3.4: g is a partial application of f and v applies both arguments.

When v applies arguments to g, it does not matter that g is defined as a partial application. This is not the case in Rust:

```rust
fn f(a: u32, b: u32) -> u32 {
    a + b
}

fn g(a: u32) -> impl Fn(u32) -> u32 {
    |x| f(a x)
}

fn v() -> u32 {
    g(1)(2)
}
```

Listing 3.5: Translation of listing 3.4 to Rust.

Note that the application of g to two arguments has become two calls. This is done by storing information about the argument splits of previously extracted functions.

This is one of multiple places where the Rust printer needs to keep state between the extraction of different declarations. This can become a problem if the declarations are provided to the printer in the wrong order. An example of this comes up in the evaluation.

3.5 Mutability and record field assignment

Record field assignment is done as described in section 2.5.

The problems surrounding interior mutability in public types made them an unappealing choice. Instead, any variable whose type is not pure is declared mut. This means any record with mutable components will become a mutable variable in Rust. This allows more mutation than WhyML, but the risk of misuse is smaller than the potential panics or undefined behavior that would come with the use of interior mutability.

3.6 Borrowing and references

References (ref) are translated directly to (mutable) values, similarly to what is done for C extraction. This works well for Copy types, but when a variable of a non-Copy type is used multiple times, the uses will get move semantics,
causing compile errors for all but the first use. Section 1.1.1 explains ownership moves and borrowing.

Another alternative is to translate ref to a Rust reference & resulting in borrows instead of moves. While using ref as an interface to Rust & could be useful as it provides a way to explicitly borrow, it would likely be difficult to use. In the 2018 Rust Survey [18], Ownership and Borrowing was one of the ‘most challenging concepts’. Problems detected during the borrow check will be even harder to understand when they refer to the extracted Rust code instead of the WhyML code that the programmer wrote.

Another solution to the problem would be for the extracted code to default to borrowing instead of moving arguments to function calls. Similarly, the driver could use borrowing to a much greater extent. This is likely to cause some code using Copy types to fail borrow checking.

3.7 Exceptions

The right way to implement this – rewriting return types to Results as discussed in section 2.10 – has many non-local effects. Exceptions are not translated, instead extraction fails when an exception is encountered.

In many cases, it would be an acceptable solution to replace exceptions with a result type in the WhyML code and use the driver to map to Rust Results. Since this solution only relies on the driver, it is easy to implement.

3.8 Formatting

Not much effort was put into making the generated Rust code readable. Instead, it is recommended to run a code formatting tool (e.g. rustfmt) on the generated code before reading.

This could be included as an automatic step during extraction, but was not to avoid dependencies on Rust binaries.

There is also an architectural reason why this feature was not included. The printer step is not fully in control of the files it generates. It is only able to generate file names and – separate from that – write one declaration at a time in the output language to a file. The architecture could be extended to add an optional file-level post-processing step, but it is not an important feature.

3.9 Derive

The evaluation doesn’t touch some of the more complex extraction cases. Type definitions is one of them. It would be useful for some extracted types to use some common derives like Clone, Copy and Debug. This is not implemented.

While it may be possible to automate the decision of whether or not to derive Clone and Copy, it would likely lead to more confusion than providing manual control.

This requires a way to provide derive annotations in WhyML. This could be implemented by adding attributes, e.g.
Listing 3.6: Suggested derive annotations

which would be handled as special cases during extraction.
Chapter 4

Evaluation

In order to evaluate the usability of the extracted code, a proven mergesort implementation distributed with the Why3 source as an example was extracted.

4.1 Overview

The file `mergesort_array.mlw` (see Appendix B) contains several different implementations of mergesort, demonstrating that some are more easily proved than others. `TopDownMergesort` splits the array in equal halves, sorts recursively and then merges. `BottomUpMergesort` sorts segments of doubling length: 1, 2, 4, ... and `NaturalMergeSort` is similar, but starts by finding sorted runs.

There are two modules that are shared by all implementations: `Merge` and `Elt`, which implement array merging and element type parametrization, respectively. The parameters `elt` and `le` are in the form of declared but undefined types and functions. `elt` is the element type and `le` is a less-than-or-equal-to predicate for the element type.

4.2 Surprise exceptions

The WhyML code for all of the variants is free from exceptions, however extraction of all but one fails with an error regarding exceptions. Closer inspection reveals that early return is represented by raising an exception in the `MLTree` representation. This is likely for the benefit of the first target language for extraction, OCaml.

It may be better to leave such language-specific transformations to the language-specific part of the extraction architecture.

For this evaluation, extraction continues with only `BottomUpMergesort`, as its `MLTree` representation contains no exceptions.

4.3 Function application bug

The solution for applying all arguments to a function that returns a closure appears to have a bug that adds an extra call in listing 4.1.
merge_using(tmp, a, lo, mid, hi);

Listing 4.1: Unnecessary double call

The return type of `merge_using` is not callable. The second pair of parentheses is removed by hand to continue the evaluation.

4.4 Array

The Why3 standard library `array.Array` module provides a rich API [5], including `sub` which takes a sub-array and `blit` which copies a run of elements between arrays. Rust arrays are not as feature-rich, but coerce to slices, which have most of the functionality needed. It is more convenient for a sorting function in Rust to take a slice, so the driver maps WhyML arrays to slices.

`blit` takes five arguments, and the mapping to Rust is not completely straightforward. The `array.Array` matches the API of the OCaml standard library and the Rust slice methods are slightly different. The driver can easily break any properties of the WhyML program by mapping incorrectly, so it is important that the driver does not take on too much responsibility and do too much work.

To make the driver creation safer, it may be desirable to adapt the (OCaml-inspired) Why3 standard library APIs to ones that are very similar to the Rust APIs that the driver will map to. Then the transformation from the OCaml-style API to the Rust API can be verified. Alternatively, a separate standard library modelling Rust can be created.

4.5 Arbitrary precision integers

The implementation uses arbitrary size integers from `int.Int` to index the arrays. This is convenient for proofs since there is no need to deal with overflowing integers. This results in much use of `BigInt` from the `num` crate in the extracted code.

Since arrays are extracted as slices, they are indexed with `usize`. This means that the driver for array has to convert the `BigInts` to potentially overflowing `usize`. As the substitutions in the driver cannot affect surrounding code, there is little that can be done in the overflow case except `unwrap` and panic. This limits the the extracted code to partial correctness: if it terminates without panic, the result is correct. This is not a problem unique to Rust; the driver for OCaml uses the potentially overflowing `Z.to_int` for array indexing.

A more interesting extraction problem comes up because `BigInt` does not implement the `Copy` trait. The Rust compiler rejects the extracted code because the slice indices are both used locally and passed along to other calls. This makes the potential problems from section 3.6 concrete.

With this in mind, the decision was made to map `int.Int` directly to finite `usize` in the driver for this extraction. This of course breaks the proof when input arrays are large enough to cause wrapping computations, but that can be seen as the program in WhyML being unsuitable for extraction to Rust. Programs that are intended to be extracted to Rust should use integer types that model the wrapping behavior of native Rust integers.
4.6 Generics

The desired Rust interface is quite different from the WhyML parametrization. The element type should be generic but bounded with a trait such as the core library’s `cmp::PartialOrd` that provides the comparison method `le`. Attempting to extract the Elt module fails because there is no definition for the parameters. Instead, a driver is specially written for this module and the module is moved to a different file to prevent it being included in a convenient file-wide extraction.

Now, the driver can map `le elt1 elt2` to `elt1.le(elt2)`, however the type `elt` is not as easy. The driver has no way to ensure it uses an unused name for the generic element type and, worse yet, there is no way to provide the trait bound. This rewrite is clearly too application-specific to merit a special case in the printer itself, but the driver is not flexible enough to provide the trait bound.

If the parametrization was changed so that it only took the comparison predicate `le 'a 'a` and left the element type generic, it would be harder to introduce the information needed for the proofs, but extraction might be easier in some ways. It would force the WhyML code to introduce a generic `'a` type in each function for the extraction to refer to. As a result, the generic type would be declared in the generated Rust, e.g. `fn merge<A> . . .`, but there would still be no way to provide a trait bound on `A`.

In order to evaluate other aspects of the extraction, the Elt driver is instead set to map to a concrete `u32` type and `<=` comparison.
Chapter 5

Discussion

We have showed that it is possible to hand-translate code, preserving meaning. The automated translation efforts are fairly successful, although some care must be taken when writing the WhyML program to avoid the more difficult translation cases. In particular, it is important to avoid using early return as it is converted into an exception before it reaches the Rust printer.

Automated translation sometimes requires analyses on the whole program, for example for better mutability decisions. The Why3 extraction framework operates on one declaration at a time, and while it is possible to build up state based on previous declarations, it is not well-supported. Looking ahead at later declarations is even harder. This limits the automated translations.

It is difficult to build good Rust interfaces in WhyML. This is important, since even if the algorithm follows the specification, a hard to understand interface can still cause it to be misused. This may be a good argument in favor of the opposite of extraction: adding contracts to Rust code and transforming to a logic language for proof.

5.1 Future work

One possible extension to this work is to automate more mappings. The mapping of exceptions is especially important.

It can also be used to inform changes to the extraction architecture, in particular to provide more context to the language-specific step and to avoid some OCaml-specific transformations that are done early in the extraction process.

Creating models for a subset of the Rust standard library in WhyML would be useful, as they can be mapped by a driver file to the native library.
Appendix A

Why3 list append

(** {1 Polymorphic Lists} *)

(** {2 Basic theory of polymorphic lists} *)

module List

type list 'a = Nil | Cons 'a (list 'a)

let predicate is_nil (l:list 'a)  
  ensures { result <-> l = Nil }
  =  
    match l with Nil -> true | Cons _ _ -> false end

end

(** {2 Length of a list} *)

module Length

use int.Int
use List

let rec function length (l: list 'a) : int =  
  match l with  
    | Nil    -> 0  
    | Cons _ r -> 1 + length r

lemma Length_nonnegative: forall l: list 'a. length l >= 0
lemma Length_nil: forall l: list 'a. length l = 0 <-> l = Nil

end

(** {2 Quantifiers on lists} *)

module Quant
use List

let rec function for_all (p: 'a -> bool) (l:list 'a) : bool =  
    match l with  
    | Nil -> true  
    | Cons x r -> p x && for_all p r  
end

let rec function for_some (p: 'a -> bool) (l:list 'a) : bool =  
    match l with  
    | Nil -> false  
    | Cons x r -> p x || for_some p r  
end

let function mem (eq:'a -> 'a -> bool) (x:'a) (l:list 'a) : bool =  
    for_some (eq x) l
end

(** {2 Membership in a list} *)

module Mem
use List

predicate mem (x: 'a) (l: list 'a) = match l with  
    | Nil -> false  
    | Cons y r -> x = y / mem x r  
end

(** {2 Appending two lists} *)

module Append
use List

let rec function (++) (l1 l2: list 'a) : list 'a =  
    match l1 with  
    | Nil -> l2  
    | Cons x1 r1 -> Cons x1 (r1 ++ l2)
end

lemma Append_assoc:  
    forall l1 [@induction] l2 l3: list 'a.  
    l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3

lemma Append_l_nil:  
    forall l: list 'a. l ++ Nil = l

use Length
use int.Int
Lemma Append_length:
  For all l1 [induction] l2: list 'a. length (l1 ++ l2) = length l1 + length l2

Use Mem

Lemma mem_append:
  For all x: 'a, l1 [induction] l2: list 'a.
  mem x (l1 ++ l2) <-> mem x l1 \ mem x l2

Lemma mem_decomp:
  For all x: 'a, l: list 'a.
  mem x l -> exists l1 l2: list 'a. l = l1 ++ Cons x l2

End
Appendix B

Mergesort

B.1 mergesort_array_params.mlw

```ml
(** {2 Parameters} *)

module Elt

use export int.Int
use export array.Array

type elt

val predicate le elt elt

clone relations.TotalPreOrder with
  type t = elt, predicate rel = le, axiom .

clone export array.Sorted with type
  elt = elt, predicate le = le, axiom .

end
```

B.2 mergesort_array.mlw

```ml
(** {1 Sorting arrays using mergesort} 
Author: Jean-Christophe Filliatre (CNRS) *)

(** {2 Parameters} *)
(* moved to separate file for easier extraction *)

(** {2 Merging} *)

It is well-known than merging sub-arrays in-place is extremely difficult (we don’t even know how to do it in linear time).
```
So we use some extra storage i.e. we merge two segments of a first array into a second array. *)

module Merge

(* clone export mergesort_array_params.EL with axiom . *)
use export mergesort_array_params.ELt
use export ref.Refint
use export array.Array
use map.Occ
use export array.ArrayPermut

(* merges tmp[l..m[ into tmp[m..r[ into a[l..r[ *)
let merge (tmp a: array elt) (l m r: int) : unit
requires { 0 <= l <= m <= r <= length tmp = length a }
requires { sorted_sub tmp l m }
requires { sorted_sub tmp m r }
ensures { sorted_sub a l r }
ensures { permut tmp a l r }
ensures { forall i: int.
(0 <= i < l / r <= i < length a) -> a[i] = (old a)[i] }
= let i = ref l in
let j = ref m in
for k = l to r-1 do
invariant { 0 <= l <= i <= m <= !j <= r }
invariant { !i - l + !j - m = k - l }
invariant { sorted_sub a l k }
invariant { forall x y: int. l <= x < k -> !i <= y < m -> le a[x] tmp[y] }
invariant { forall x y: int. l <= x < k -> !j <= y < r -> le a[x] tmp[y] }
invariant { forall v: elt.
occ v tmp.elts l !i + occ v tmp.elts m !j = occ v a.elts l k }
invariant { forall v: elt.
occ v tmp.elts !i + occ v tmp.elts !j = occ v a.elts l k }
if !i < m && (!j = r || le tmp[!i] tmp[!j]) then begin
  a[k] <- tmp[!i];
  incr i
end else begin
  a[k] <- tmp[!j];
  incr j
end
done

(* merges a[l..m[ into a[m..r[ using tmp as a temporary *)
let merge_using (tmp a: array elt) (l m r: int) : unit
requires { 0 <= l <= m <= r <= length tmp = length a }
requires { sorted_sub a l m }
requires { sorted_sub a m r }
ensures { sorted_sub a l r }
ensures { permut (old a) a l r }
ensures { forall i: int.
(0 <= l <= m && m <= r <= length a) -> a[i] = (old a)[i] }
= if l < m && m < r then (* both sides are non empty *)
  if le a[m-1] a[m] then (* OPTIM: already sorted *)
    assert { forall i1 i2: int. l <= i1 < m -> m <= i2 < r ->
    ...

32
le a[i1] a[m-1] && le a[m] a[i2] }
else begin
  label N in
  blit a l tmp l (r - l);
  merge tmp a l m r;
  assert { permut_sub (a at N) a l r }
end

end

(** {2 Top-down, recursive mergesort}
Split in equal halves, recursively sort the two, and then merge. *)

module TopDownMergesort

clone Merge with axiom .
(* use Merge *)
use mach.int.Int

let rec mergesort_rec (a tmp: array elt) (l r: int) : unit
  requires { 0 <= l <= r <= length a = length tmp }
  ensures { sorted_sub a l r }
  ensures { permut_sub (old a) a l r }
  variant { r - l }
  = if l >= r-1 then return;
  let m = l + (r - l) / 2 in
  assert { l <= m < r };
  mergesort_rec a tmp l m;
  assert { permut_sub (old a) a l r };
  label M in
  mergesort_rec a tmp m r;
  assert { permut_sub (a at M) a l r };
  merge_using tmp a l m r

let mergesort (a: array elt) : unit
  ensures { sorted a }
  ensures { permut_all (old a) a }
  =
  let tmp = Array.copy a in
  mergesort_rec a tmp 0 (length a)

end

(** {2 Bottom-up, iterative mergesort}
First sort segments of length 1, then of length 2, then of length 4, etc. until the array is sorted.
Surprisingly, the proof is much more complex than for natural mergesort (see below). *)

module BottomUpMergesort
(* clone Merge with axiom . *)

use Merge
use mach.int.Int
use int.MinMax

let bottom_up_mergesort (a: array elt) : unit
  ensures { sorted a }
  ensures { permut_all (old a) a }
  = let n = length a in
  let tmp = Array.copy a in
  let len = ref 1 in
  while !len < n do
    invariant { 1 <= !len }
    invariant { permut_all (old a) a }
    invariant { forall k: int. let l = k * !len in
               0 <= l < n -> sorted_sub a l (min n (l + !len)) }
    variant { 2 * n - !len }
    label L in
    let lo = ref 0 in
    let ghost i = ref 0 in
    while !lo < n - !len do
      invariant { 0 <= !lo /
                  !lo = 2 * !i * !len }
      invariant { permut_all (a at L) a }
      invariant { forall k: int. let l = k * !len in
                  !lo <= l < n -> sorted_sub a l (min n (l + !len)) }
      invariant { forall k: int. let l = k * (2 * !len) in
                  0 <= l < !lo -> sorted_sub a l (min n (1 + !len)) }
      variant { n + !len - !lo }
      let mid = !lo + !len in
      assert { mid = (2 * !i + 1) * !len };
      assert { sorted_sub a !lo (min n (!lo + !len)) };
      let hi = min n (mid + !len) in
      assert { sorted_sub a mid (min n (mid + !len)) };
      label M in
      merge_using tmp a !lo mid hi;
      assert { permut_sub (a at M) a !lo hi };
      assert { permut_all (a at M) a };
      assert { hi = min n (!lo + 2 * !len) };
      assert { sorted_sub a !lo (min n (!lo + 2 * !len)) };
      assert { forall k: int. let l = k * !len in
               mid + !len <= l < n ->
               sorted_sub (a at M) l (min n (l + !len)) &
               sorted_sub a l (min n (l + !len)) };
      assert { forall k: int. let l = k * (2 * !len) in
               0 <= l < mid + !len ->
               k <= !i &
               (k < !i ->
                min n (l + 2 * !len) <= !lo &
                sorted_sub (a at M) l (min n (l + 2 * !len)) &
                sorted_sub a l (min n (l + 2 * !len)) ) &
               (k = !i ->
                l = !lo /
                sorted_sub a l (min n (1 + 2 * !len))) };
      lo := mid + !len;
      ghost incr i
done;
assert { forall k: int. let l = k * (2 * !len) in 0 <= l < n ->
  l = (k * 2) * !len &&
  (l < !lo ->
    sorted_sub a l (min n (l + 2 * !len)) &&
    (l >= !lo ->
      sorted_sub a l (min n (l + !len)) &&
      min n (l + 2 * !len) = min n (l + !len) = n )
  )
};
len := 2 * !len;
done;
assert { sorted_sub a (0 * !len) (min n (0 + !len)) }
end

(** {2 Natural mergesort}
This is a mere variant of bottom-up mergesort above, where
we start with ascending runs (i.e. segments that are already sorted)
instead of starting with single elements. *)

module NaturalMergesort
clone Merge with axiom .
(* use Merge *)
use mach.int.Int
use int.MinMax

(* returns the maximal hi such that a[lo..hi[ is sorted *)
let find_run (a: array elt) (lo: int) : int
requires { 0 <= lo < length a }
ensures { lo < result <= length a }
ensures { sorted_sub a lo result }
ensures { result < length a -> not (le a[result-1] a[result]) }
=
  let i = ref (lo + 1) in
  while !i < length a && le a![i - 1] a![i] do
    invariant { lo < !i <= length a }
    invariant { sorted_sub a lo !i }
    incr i
  done;
  !i

let natural_mergesort (a: array elt) : unit
ensures { sorted a }
ensures { permut_all (old a) a }
let n = length a in
if n <= 1 then return;
let tmp = Array.copy a in
let ghost first_run = ref 0 in
while true do
  invariant { 0 <= !first_run <= n && sorted_sub a 0 !first_run }
  invariant { permut_all (old a) a }

end
variant { n - !first_run }

label L in

let lo = ref 0 in

while !lo < n - 1 do
  invariant { 0 <= !lo <= n }
  invariant { !first_run at L <= !first_run <= n }
  invariant { sorted_sub a 0 !first_run }
  invariant { !lo = 0 / !lo >= !first_run > !first_run at L }
  invariant { permut_all (a at L) a }
  variant { n - !lo }
  let mid = find_run a !lo in
  if mid = n then begin if !lo = 0 then return; break end;
    let hi = find_run a mid in
    label M in
    merge_using tmp a !lo mid hi;
    assert { permut_sub (a at M) a !lo hi };
    assert { permut_all (a at M) a };
    ghost if !lo = 0 then first_run := hi;
    lo := hi;
  done

variant { n - !lo }

let mid = find_run a !lo in
if mid = n then begin if !lo = 0 then return; break end;
  let hi = find_run a mid in
  label M in
  merge_using tmp a !lo mid hi;
  assert { permut_sub (a at M) a !lo hi };
  assert { permut_all (a at M) a };
  ghost if !lo = 0 then first_run := hi;
  lo := hi;
  done

/** an alternative implementation suggested by Martin Clochard,
  mixing top-down recursive and natural mergesort 

  the purpose is to avoid unnecessary calls to [find_run] in 
  the code above *)

let rec naturalrec (tmp a: array elt) (lo k: int) : int
  requires { 0 <= lo <= length a = length tmp }
  requires { 0 <= k }
  ensures { result = length a \ / lo + k < result < length a }
  ensures { sorted_sub a lo result }
  ensures { permut_sub (old a) a lo (length a) }
  ensures { forall j: int. 0 <= j < lo -> a[j] = (old a)[j] }
  variant { k }
  = let n = length a in
  if lo >= n-1 then return n;
  let mid = ref (find_run a lo) in
  if !mid = n then return n;
  for i = 0 to k-1 do
    invariant { lo + i < !mid < n }
    invariant { sorted_sub a lo !mid }
    invariant { permut_sub (old a) a lo (length a) }
    invariant { forall j: int. 0 <= j < lo -> a[j] = (old a)[j] }
    let hi = naturalrec tmp a !mid i in
    assert { permut_sub (old a) a lo (length a) };
    label M in
    merge_using tmp a lo !mid hi;
    assert { permut_sub (a at M) a lo hi };
    assert { permut_sub (a at M) a lo (length a) };
    mid := hi;
    if !mid = n then return n
```haskell
let natural_mergesort2 (a: array elt) : unit
  ensures { sorted a }
  ensures { permut_all (old a) a }
  =
    let tmp = Array.copy a in
    let _ = naturalrec tmp a 0 (length a) in
() end

B.3 mergesort_rust.drv driver

module mergesort_array_params.Elt
  syntax type elt "u32"
  syntax val le ".__val__le"
  syntax predicate le "%.u.<=%.u2"
end

B.4 mergesort.rs extracted code

pub fn merge(
  tmp: &mut [u32],
  a: &mut [u32],
  l: usize,
  m: usize,
  r: usize,
) -> () {
  let mut i = { l };  
  let mut j = { m };  
  let o = { r - 1 };  
  for k in l..=o {
    if if i < m {
      if j == r {
        true
      } else {
        tmp[i] <= tmp[j]
      }
    } else {
      false
    }
  }
}
```
pub fn merge_using(
    tmp: &mut [u32],
    a: &mut [u32],
    l: usize,
    m: usize,
    r: usize,
) -> () {
    if if l < m { m < r } else { false } {
        if a[m - 1] <= a[m] {
            {}
        } else {
            { tmp[l..(l + r - l)].clone_from_slice(&a[l..(l + r - l)]);
                merge(tmp, a, l, m, r)
            }
        }
    } else {
        {}
    }
}

pub fn bottom_up_mergesort(a: &mut [u32]) -> () {
    let n = { a.len() };
    let mut tmp = {
        &mut a.iter().cloned().collect::<Vec<u32>>()[..]
    };
    let mut len = { 1 };
    while len < n {
        let mut lo = { 0 };
        { while lo < n - len {
            let mid = { lo + len };
            let hi = { n.min(mid + len) };
            { merge_using(tmp, a, lo, mid, hi);
                lo = mid + len
            }
        }
        len = 2 * len
    }
}
Bibliography


