Vrije Universiteit Amsterdam Universiteit van Amsterdam

Master Thesis

Correctness proofs for C library functions with Frama-C

Author: Vladislav Yaglamunov (2672631)

1st supervisor:	W.J. Fokkink	
daily supervisor:	M. Beemster	(Solid Sands)
2nd reader:	G. Ebner	

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 12, 2021

Abstract

This work describes human guided formal verification of C library functions. For this purpose, the Frama-C toolkit was used. Several presented techniques are highly specific to the used toolkit, nevertheless, a number of techniques can be used in other analogous systems. The thesis focuses primarily on two function types: string and floating-point function. For the former type, a simple and a replicable method of verification is created, the crucial benefit of which is full automation of proofs. This makes the process effortlessly accessible to actual C developers. The floating-point function proved to be a significantly more complex task. Even though a similar full automation for them was not achieved, several techniques were developed to partially simplify and streamline its verification.

Contents

1

Introduction

This thesis aims to explore methods of formal verification of the C language. Functions from the standard C library are formally verified. Using functions from the C library allows to verify them against their standard description. In order to do so, a human language specification of the functions must be translated into a formal language definition. This could sometimes be a non-trivial task because as any prose style natural language text, the C standard suffers from ambiguities and common-sense implications.

The goal is to develop several approaches to C standard translation into a formal specification and its further verification. One of the focuses is to create verification methods that a C developer can apply without profound knowledge of formal methods.

For this reason, the Frama-C toolkit was chosen as a human-guided verification system. It was in active development in recent years, used in multiple projects mentioned later. Importantly, this toolkit is one of the tools that is close to actual C, which allows it to be used by a proficient C developer without too much extra knowledge or training.

The standard C library contains a vast variety of function and verifying it fully is out of the scope for this project. Two different types of functions were selected for this work. The first is string manipulation functions, these functions focus on pointer operations and memory accesses. The second is floating-point functions. Two selected types allow testing formal methods in significantly different environments.

The string functions rely only on core principles of the language. They don't require any special system calls and usually don't have any side effects. Furthermore, they feature a heavy use of loops and pointer manipulations, which is also very common in general C programs. This makes them a good starting point to try how feasible is this project.

The floating-point functions also mostly contain no system calls or side effects. However, they provide a significantly greater challenge. A lot of critically important code used in

automotive or avionics industries heavily rely on math library. Presenting an accessible way to formally verify functions from the math library would substantially aid to make this code more safe and trustworthy.

1.1 Formal C verification

Formal verification is the process of using mathematical methods to prove compliance or non-compliance of the subject with its formal description. The formal subject could be an algorithm, a program or another proof. There are many approaches to formal verification, ranging from fully automated ones that are usually meant to only guarantee the absence of run-time errors and undefined behaviors to approaches with human-guidance that will guarantee full functional correctness.

The notion of formal verification was introduced by Floyd and Hoare in the 1960s (1, 2). They proposed an axiomatic system, that nowadays is called a *Hoare logic*. This system allows describing computer program properties in mathematic language and reason about them. Following on Floyd and Hoare ideas, Dijkstra (3) invented the concept of the *weakest preconditions*. This concept became the base of formal verification with condition generation. It takes a program with logical specification annotations and produces a set of verification conditions. If it is proved that all conditions are met, the program is guaranteed to behave correctly.

1.2 Formalization of C language

One way to reason about C programs is to create a formal environment with an explicit semantics, this can be achieved in various proof assistants such as Coq (4), Isabelle (5), or HOL4 (6). The proof assistant can be used to formulate theoretical properties of the language and then to verify them on actual C programs.

The formalized semantics of the language allow to have a completely precise standard without inconsistencies of human prosaic language. This is incredibly useful for writing and verifying compilers. However, it is also useful for any C code verification, it allows establishing properties of a C programs and make an explicit proven connection from this property to the official C standard.

The first formalized C semantics has been developed by Norrish using HOL4 language to formalize selected parts of the C89 standard (7). Later, C formalized semantics were used

in compiler verification. Leroy used C semantics formalized in Coq to prove correctness of optimizations in the CompCert compiler (8). CompCert is written in Coq itself.

The more recent development in this area was made by Krebbers in his CH_2O project that aims to formalize C standard in Coq (9). The goal of the project was to develop a formal version of the non-concurrent fragment of the C11 standard that is usable in proof assistants. It provides Operational, Executable and Axiomatic semantics for C in Coq. It also verifies that all semantics correspond to each other.

1.3 Human assisted verification

However, it is usually unfeasible to use this formalized semantics to verify any industrial code due to its volume and complexity. Additionally, usage of it requires profound knowledge of formal logic and the used proof assistant language. This makes it unaccessible to developers of the program, as they usually are proficient in different areas.

In contrast to formalization approaches, human-guided systems make a less explicit connection to the standard and relay more on generated logical conditions. But the simplicity and efficiency of use allows them to verify more practical programs with less specific knowledge of formal methods.

These systems are mostly based on a form of verification condition generation, where human guidance is required to annotate the given program with logical conditions. The resulting verification conditions are then verified either fully automatically using SAT/SMT solvers, or interactively using proof assistants. Examples of such systems are Boogie (10), HAVOC (11), the Jessie and WP plug-ins of Frama-C (12), Key-C (13), VCC (14) and Verifast (15)

1.4 Frama-C toolkit and WP plugin

This work focuses primarily on verification with Frama-C as one of the tools with still active development. Frama-C is an open-source platform and collection of tools for analysis of source code written in the C programming language. The platform is developed with collaboration from two public French institutions, CEA LIST and Inria Saclay. The toolkit contains multiple different plugins, the main goal is providing correct static analysis, but it also includes plugins for source code navigation, code transformations and reporting.

The WP plugin from the toolkit is of special interest to this work. The plugin allows for deductive proofs of function contracts. It is based on weakest-precondition calculus and

relies on external theorem provers to finalize the assessment of the desired properties (16). Here, mostly Alt-Ergo (17) and Z3 (18) theorem provers are used. Both are open-source solvers based on Satisfiability Modulo Theories (SMT).

When automated provers cannot verify some part of the contract, an external proof assistant can be used. Coq, the popular proof assistant supported by the WP plugin, is used in this work. Since the goal is to make human-guided formal verification more accessible to C developers, an extra effort was put to avoid the use of a proof assistant. When it is not possible, an attempt was made to minimize required knowledge and simplify the process of applying it.

The example of using Frama-C with the Jessie plug-in for formal verification of industrial code for Dassault Aviation was shown in (19). The paper shows some common difficulties and solutions when working with such systems, focusing heavily on automation of the process. Noticing that manually annotating every function in the source code is a fastidious and costly task with a risk of human error, it proposes an extra plugin for the Frama-C framework that could generate simple annotations. The second automation was to translate Frama-C lemmas into Maxima (20, 21) hypotheses and goals instead of manually proving them in Coq.

1.5 Floating-point verification

Floating-point applications, such as simulations in nuclear physics or aeronautics, often require a high level of guarantee. However, it could be difficult to achieve due to the nature of floating-point operations that involve multiple roundings and potential exceptions, as shown in (22). Consequently, it is hard to describe denotationally in a logic setting.

Despite that, formal methods were successfully applied to verify a floating-point arithmetic in the IMS T800 Transputer processor in 1989 (23). A first proposal to verify a floating-point C program has been made in 2007 (24) using Caduceus (25) (a predecessor of the Frama-C toolkit) and an existing formalization of floating-point arithmetic in Coq.

1.6 Contributions

This work primary focuses on using human-guided systems (Frama-C in particular) to verify C library functions. The objective is to make this process robust and accessible. One step to achieve these qualities is to minimize the amount of proof that had to be carried manually using external proof assistants like Coq.

For the first selected library section on string and array functions (Chapter 2) methods have been developed to verify these functions fully automatically requiring only code annotations. Feasibly, the demonstrated approach can be effortlessly replicated by C developers and does not require special formal verification training.

For the second section on floating-point (Chapter 3) the thesis discusses reasons why the currently existing tools are not developed enough to make verification broadly accessible. Furthermore, a few approaches are shown to potentially slightly simplify the process.

The source code for the examples used in the work and materials in the Appendix are available on-line at `https://github.com/VladYagl/C-library-verification`.

2

String functions, memory and pointer manipulation

2.1 Background

A function contract consists of two parts: preconditions and postconditions.

A *precondition* is a property about the input of the function or about global state and variables before the function call. This property is assumed to be always true when the function is called. It is a part of a function contract that a caller must obey in order to use the function. If the precondition does not hold when the function is called, its behavior is undefined.

A *postcondition* of a function is a property about the output of the function or the global variables after the function. This property must hold when the function successfully terminates. It is a part of a function contract that the function itself obeys.

A function contract is considered to be verified if it is proved that if before the function call all its preconditions were met then the function will always terminate successfully with all its postconditions holding.

This work showcases different approaches to translating a standard description of a C library function to a formal contract and verifying such a contract. For defining contracts, the ANSI/ISO C Specification Language (ACSL) is used. For proving ACSL contracts, a WP plugin of the Frama-C toolkit is used.

2.2 Verifying pointer operations

2.2.1 Pointer loop

A very frequent pattern in a string manipulating functions is traversing through an array by incrementing a single pointer (e.g. `for (; *s; s++);`). In order to verify such a loop, this is important to provide Frama-C with some information about the pointer using loop invariants.

The common practices for pointer incrementing loops are described in an example of a `strlen` function implementation from the `musl` C library:

```
size_t strlen(const char *s)
{
    const char *a = s;
    for (; *s; s++);
    return s-a;
}
```

Before verifying any function, it is a must to strictly formulate what exactly is being verified. In the WP plugin of Frama-C it is done by creating a function contract with pre- and postconditions of the function.

The `strlen` description is "The strlen function computes the length of the string pointed to by s." In the standard C library, string length is defined as the "number of characters that precede the terminating null character". However, explicitly calculating this value in logical expressions of ACSL is not possible because the string might not contain a terminating null character. Such a string is technically of an "infinite" length. A good way to separate this case would be to define that "infinite" strings have a negative length.

Because of this, the axiomatic logical *strlen* function was created. A logical function is a function defined in ACSL that is meant to describe functions that can only be used in logical expressions like contract definitions. And an axiomatic logical function is a function that does not have an explicit definition of the function, but only has axioms about the function.

2.2.2 Logical strlen function

```
axiomatic StrLen {
    logic ℤ strlen(char* s) reads s[0 .. ];

    axiom pos_or_nul:
        ∀ char* s, ℤ i;
            (0 ≤ i ∧
            (∀ ℤ j; 0 ≤ j < i ⇒ s[j] ≢ '\0') ∧
            s[i] ≡ '\0') ⇒
                strlen(s) ≡ i;

    axiom no_end:
        ∀ char* s; (∀ ℤ i; 0 ≤ i ⇒ s[i] ≢ '\0') ⇒
            strlen(s) < 0;

    axiom less_then_size_t:
        ∀ char* s; strlen(s) ≤ SIZE_T_MAX;
}
```

Preceding the axioms is an abstract definition of logical *strlen*. Apart from specifying output and input types of the function it also defines what is read by the function. In this case it is all memory after the given pointer until an abstract infinite address.

The first two axioms define core properties of the *strlen* function. `pos_or_null` describes a case of a valid string: if the ith symbol is null (`\0` is an escape sequence of the null symbol, it also has value of 0) and there is no null symbol before i then $strlen = i$. And `no_end` describes a case when there is no null symbol in the string, so *strlen* is negative. The last axiom limits the maximum length of a string. It is needed so that is possible to verify that a `size_t` integer variable can be assigned a length of any string without an overflow.

Before moving onto validating actual functions, a few lemmas are added to help reasoning about logical *strlen*:

```
lemma index_of_strlen:
    ∀ char* s; strlen(s) ≥ 0 ⇒ s[strlen(s)] ≡ '\0';

lemma before_strlen:
    ∀ char* s; strlen(s) ≥ 0 ⇒
        (∀ ℤ i; 0 ≤ i < strlen(s) ⇒ s[i] ≢ '\0');

lemma neg_len:
    ∀ char* s; strlen(s) < 0 ⇒
    (∀ ℤ i; 0 ≤ i ⇒ s[i] ≢ '\0');
```

These lemmas define reversed implications from core axioms of *strlen*. The first two lemmas express that if *strlen(s)* is nonnegative, then there is a null symbol at *strlen(s)* and there are no null symbols before it. On the contrary, the third lemma states that there is no null symbol after the string start if *strlen(s)* is negative. In this case, the string is considered invalid.

Finally, the next lemma states that if strings are equal then their lengths are equal, too:

```
lemma same:
    ∀ char *s, *d; strlen(s) ≥ 0 ⇒
        (∀ ℤ i; 0 ≤ i ≤ strlen(s) ⇒ s[i] ≡ d[i]) ⇒
            strlen(s) ≡ strlen(d);
```

The lemmas cannot be proved automatically because the proof requires reasoning by induction. But due to the fact that these lemmas are reversed properties of axiomatic definition, if it is proven that the initial conditions of the axioms are mutually exclusive then the automatic prover can finish the job. For `pos_or_null` the condition is: i is a first null symbol in the string; and for `no_end` it is: there are no null symbol in the string. So, it is needed to prove a simple lemma that if there is a null symbol in the string, then there is the first null symbol in the string:

```
lemma exists_first:
    ∀ char* s; (∃ ℤ i; 0 ≤ i ∧ s[i] ≡ '\0') ⇒
        (∃ ℤ i; (0 ≤ i ∧
            (∀ ℤ j; 0 ≤ j < i ⇒ s[j] ≢ '\0') ∧
            s[i] ≡ '\0'));
```

The proof of the lemma is constructed using Coq. A full description of it can be found in Appendix A.2. With the `exists_first` lemma proved, all original four lemmas are proved automatically by the WP plugin.

Further, in this work, to distinguish between logical and C-language string length functions: *strlen* will be used for the logical function and `strlen` will be used for the C function.

2.2.2.1 Function contract

Now, using the logical *strlen* function, it is possible to specify a function contract for the C language `strlen` function:

```
/*@
    requires strlen(s) ≥ 0 ∧ \valid_read(s + (0 .. strlen(s)));
    assigns \nothing;
    ensures \result ≡ strlen(s);
*/
size_t strlen(const char *s);
```

This contract showcases three different parts of any function contract in ACSL.

The first is the `requires` part. This keyword is used to specify preconditions of the function. In this case, the only precondition is for the input string to be a valid string to read. The requirement on a valid read comes from the paragraph 7.1.4 "Use of library function" of the C standard :

> If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid.

To check this property, it is needed to understand what memory the function needs to access. For `strlen` it accesses all memory from the given pointer until the null symbol. So, it goes through all pointers from `s[0]` until `s[strlen(s)]`. In ACSL, the range of pointers is defined using the `".."` symbol and the `\valid_read` function is used to check that the pointer can be dereferenced, but only to read the pointed memory. Moreover, the string must be finite, thus logical *strlen* must not be negative.

Second is the `assigns` part. It is used to specify side effects of the function. The `strlen` function does not change any global variable or write to any global memory. So `strlen` does not have any side effects to express that a special keyword `\nothing` is used.

The last part is the `ensures` which is used to declare postconditions of the function. For `strlen` the postcondition is that the return value of the function is equal to the previously defined logical *strlen* function.

After the contract is defined, the WP plugin usually cannot prove it automatically, and it is needed to aid it by providing meaningful asserts and loop invariants inside the function. When the contract is defined, it is finally possible to move to verifying the pointer loop inside the function.

2.2.2.2 Helper predicates

The valid string properties used in the `requires` part of the `strlen` function will be very common, so a few predicates to define similar properties are created.

Frama-C provides two helpful predicates to reason about valid pointers: `valid` and `valid_read`. The first requires a pointer to be valid to read and write, the second only requires this for reads. The same pattern can be applied to string validity. A valid string s is valid from its first index to *strlen(s)*, where this value is nonnegative (since an infinite string is not a valid string):

```
predicate valid_read_string(char* s) =
    strlen(s) ≥ 0 ∧ \valid_read(s + (0 .. strlen(s)));

predicate valid_string(char* s) =
    strlen(s) ≥ 0 ∧ \valid(s + (0 .. strlen(s)));
```

Additionally, a predicate to compare two strings is defined. Two strings are considered equal if they have the same length, and all symbols until the length are identical.

```
predicate array_equal{L1, L2}
    (char* a, ℤ a_begin, char* b, ℤ b_begin, ℤ len) =
    ∀ ℤ i; 0 ≤ i < len ⇒
    \at(a[a_begin + i], L1) ≡ \at(b[b_begin + i], L2);

predicate array_equal{L1, L2}(char* a, char* b, ℤ len) =
    array_equal{L1, L2}(a, 0, b, 0, len);

predicate string_equal{L1, L2}(char* a, char *b) =
    \at(strlen(a), L1) ≡ \at(strlen(b), L2) ∧
    array_equal{L1, L2}(a, b, strlen{L1}(a) + 1);
```

2.2.3 Memory models in WP

The WP plugin has two distinct memory models built into it.

The first one is the *Hoare Model*, a very simple and efficient model for concise proof obligations. It works by simply mapping individual C variables to one separate pure logical variable. However, this model has a significant limitation: memory accessing operations cannot be translated at all (such as `*p`) (16). This makes this model not very practical for string manipulation functions, as they heavily rely on reading and writing arrays in memory.

The second one is the *Typed Model*, which is also a default model for Frama-C. In this model, heap values are separated by a primitive type. For each type, there is a global array that stores values of this type. Pointers are then translated into indexes in these arrays (16). This model allows reasoning about operations on memory, and as such is more suited for string functions.

To simplify reasoning about pointer separation and commutative memory accesses, pointers are not modeled as a single integer index in the array but as a pair of two integers (base address and offset). So for two pointers to be equal both base and offset addresses must be equal and both must be of the same type (for example unsigned 8-bit integer).

2.2.3.1 Pointer base.

Returning back to the `strlen` function. The internal loop of it cannot be verified automatically. The WP plugin requires the programmer to provide loop invariants for each loop inside of the function. The invariant must hold before the loop, and if it held before the loop step, it should hold after it.

However, invariants do not guarantee a loop termination. To verify it one must provide a positive value that strictly decreases in each loop step. Because a positive value cannot decrease infinitely, the existence of a such value proves that a loop terminates. In WP, this value is called `loop variant`.

The first step is to make an invariant that the base part of the pointer stays the same during the loop and only the offset part changes. However, the ACSL syntax does not allow to reason about values of a base and an offset of a pointer directly. To work around that, a simple predicate is made:

```
predicate based(unsigned char* a, unsigned char* b) =
    a + (b - a) ≡ b;
```

At a first glance, this predicate might seem trivial: there is `a` and `-a` on the left side, so the expression could be simplified to `b` ≡ `b`. However, pointers are actually modeled as pairs of integers. Inspecting the left side in the "WP Goals" tab of Frama-C shows

that it translates to `shift(a_0, b_0.offset - a_0.offset)`. And after applying "Qed" simplifications, the whole predicate becomes `b_0.base = a_0.base`, which is an original goal of this predicate.

To apply this predicate on the same pointer but at different points in time during program execution, the overloaded version of the predicate is used:

```
predicate based{L1, L2}(char** a) =
    based(\at(*a, L1), \at(*a, L2));
```

It is important to use a pointer to a pointer. The reason for it is that the goal is to reason about the value of the pointer at different labels during execution. The built-in `\at` function allows the user to basically read memory at different program states captured on C labels. If a normal pointer would be used, the value of the pointer would be captured once at the moment of calling the predicate and thus yield an identical value.

Using predicates defined above, an invariant for a `strlen` function loop will look like this (`Pre` and `Here` are labels predefined by the WP plugin referring to the states before function call and at the current moment respectively):

```
loop invariant based: based{Pre, Here}(&s);
```

2.2.3.2 Limit index.

The next step is to create an index for a loop. Currently, the loop doesn't have a clear index. It increments the pointer until it points at null symbol. Usually, the difference between the starting position of a pointer and its current position could be such an index: `s - \at(s, Pre)`.

After deciding on an index, one needs to limit its possible values and show that it will eventually reach one of the limits. For the `strlen` function, the index starts at 0 and goes up to a value of the logical *strlen* function:

```
loop invariant under_len:
    0 ≤ s - \at(s, Pre) ≤ \at(strlen(s), Pre);

loop variant \at(strlen(s), Pre) - (s - a);
```

The `loop invariant` here provides limits for an index, and the `loop variant` shows that the index strictly increases and will eventually reach its maximum.

2.2.3.3 Memory correlation

With all the provided invariants, Frama-C can deduce that after the loop, the value of the previously defined "virtual" index is between 0 and the value of the logical *strlen*. And that pointer `s` is currently at the null symbol. It is possible to observe that by adding two asserts after the loop:

```
//@ assert *s ≡ 0;
//@ assert 0 ≤ s - a ≤ \at(strlen(s), Pre);
```

Both asserts are proven automatically by "Qed". However, this knowledge is still not enough for the automatic prover to verify the postcondition of the `strlen` function. It is missing the connection between the original memory pointed by `s` and the current value of `*s`. After adding the assert after the loop that states exactly that, the post condition of `strlen` is finally proven automatically by the WP plugin.

```
//@ assert (\at(s, Pre))[s - \at(s, Pre)] ≡ *s;
```

It is easy to notice similarity between the last assert and the initial *based* predicate. The memory correlation problem is also quite common for pointer loops. For these reasons, the last assert is extracted as a separate predicate, again very similar to the *based* one, it differs only by comparing memory at the pointers and not the pointers themselves:

```
predicate based_ptr(char *a, char* b) =
    *(a + (b - a)) ≡ *b;
```

And identically to *based* this predicate can be overloaded to take a single pointer at different moments in time. Now the assert can be replaced using the newly created predicate. However, one can observe that this property holds for each loop step. This makes it usually more efficient to have it as a loop invariant. It allows to reason about pointed memory inside the loop, which is very useful in more complicated examples:

```
loop invariant based_ptr: based_ptr{Pre, Here}(&s);
```

Now the final assert can be removed in favor of the loop invariant above. The postcondition of `strlen` is still proven automatically by Frama-C.

The combination of the two created predicates (*based* and *based_ptr*) as loop invariants is very useful when working with loops traversing memory.

Both hold when the base of the pointer is unchanged, i.e. the pointer was only modified via arithmetics like increments or decrements and was not assigned a new value. For simple cases (like with the `strlen` function) providing two of them might be enough for the WP plugin to finish verification automatically.

2.2.4 strncat

This section shows the application of the concepts shown before on a more complicated example of a string concatenation function. The musl library implementation of the strncat function is very concise and contains a single one-line while loop:

```
#include <string.h>

char *strncat(char *restrict d, const char *restrict s, size_t n)
{
        char *a = d;
        d += strlen(d);
        while (n && *s) n--, *d++ = *s++;
        *d++ = 0;
        return a;
}
```

However, there are multiple actions happening in this line. Frama-C expands the loop to make every action defined explicitly in a single operation. It also splits the loop condition in separate if-break statements.

```
while (1) {
    char *tmp_0;
    char const *tmp_1;
    if (n) { if (! *s) { break; } }
    else { break; }
    n --;
    { /* sequence */
        tmp_0 = d;
        d ++;
        tmp_1 = s;
        s ++;
        *tmp_0 = *tmp_1;
    }
}
{ /* sequence */
    tmp_2 = d;
    d ++;
    *tmp_2 = (char)0;
}
```

Approaching this loop might seem a daunting task. Nevertheless, using the previously defined principles makes providing loop invariants for it very straightforward.

2.2.4.1 Function contract

The logical contract for a function must first and foremost comply with the C standard definition of the function. For `strncat` the standard says:

> 'The strncat function appends not more than n characters (a null character and characters that follow it are not appended) from the array pointed to by s2 to the end of the string pointed to by s1. The initial character of s2 overwrites the null character at the end of s1. A terminating null character is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined.
>
> **Returns** The strncat function returns the value of s1.'

Also, as before for `strlen` from paragraph 7.1.4 of the standard, string pointers are assumed to be valid strings. Which is *strlen(s)* is positive and all pointers from initial

untill *strlen* are valid pointers. This can be checked using the helper predicate that was defined earlier. This gives a first requirement for the `strncat` function. The destination string must be a valid string to read.

A validity of the source string is slightly trickier. There are two possible cases: either `strncat` will stop coping with symbols upon reaching the nth symbol or upon reaching the end of the string (null symbol). In the first case, the first n addresses after the initial pointer are required to be valid to read. In the second case, the whole source string is required to be a valid string. And since the source string is appended after the destination, the addresses it is appended to must also be valid to write. They follow exactly the same logic: either it is first n symbols after the destination end if the string is copied partially, or first *strlen(s)* if the string is copied fully.

The `strncat` has two slightly different behaviors depending on the input. This can be expressed using the `behavior` keyword:

```
behavior big_enough:
    assumes big_enough: ((n ≤ strlen(s)) ∨ (strlen(s) < 0))
    requires valid_src: \valid_read(s + (0 .. (n - 1)));
    requires valid_dest: \valid(d + strlen(d) + (0 .. n));
    ...

behavior small:
    assumes small: 0 ≤ strlen(s) < n;
    requires valid_src: valid_read_string(s);
    requires valid_dest: \valid(d + strlen(d) + (0 .. strlen(s)));
    ...
```

The first behavior assumes that string length is greater than n or the string is not valid (a plain `char` array was given). In this case, it is enough to have valid source and destination memory segments of length n (no need for the `valid_string` function). The second one assumes the *strlen* to be valid (nonnegative) and smaller than n. Then the source string is required to be valid, and the destination is valid to write for *strlen(s)* number of characters.

However, this approach introduces extra complexity in the verification process. For each logic statement inside of a function (usually asserts or loop invariants) two separate goals will be generated: one for each behavior, thus generating twice as many goals. It slows down proof generation and when using iterative scripts for some goals (Coq or built-in script) potentially one will have to write the same script twice for both behaviors.

This can be avoided by extracting this varying behavior into a separate logic function. This function will calculate the number of copied symbols from the source, depending on the source's *strlen* and the value of n:

```
logic ℤ min_len(ℤ len, ℤ n) = (0 ≤ len < n) ? len : n;

logic ℤ len(char* s, ℤ n) = min_len(strlen(s), n);
```

Using this function `strncat` has 4 requirements:

- the destination string is a valid string to read (`valid_read_string`)
- the source string is valid to read until `len(s)`.
- the destination string is valid to write `len(s)` symbols starting from its end.
- strings do not overlap with each other.

The first 3 follow from a requirement on a valid pointer accesses (paragraph 7.1.4 of the standard). The last one follows from the definition of a function ("If copying takes place between objects that overlap, the behavior is undefined.")

In ACSL, these requirements can be defined in the following way:

```
requires valid_d: valid_read_string(d);
requires valid_src: \valid_read(s + (0 .. (len(s, n - 1)) ));
requires valid_dest: \valid(d + strlen(d) + (0 .. len(s, n)));
requires separation: \separated(
        &d[0 .. (strlen(d) + len(s, n))],
        &s[0 .. len(s, n - 1)]
    );
```

Now moving to postconditions. One obvious postcondition would come from a main definition of `strncat`: "appends not more than n characters (a null character and characters that follow it are not appended) from the array pointed to by s2 to the end of the string pointed to by s1". The "not more" part is exactly what is modeled by the `len` function, and appended means that the next `len(s)` symbols after the destination must be equal to the first `len(s)` symbols of the source string. The helper predicate is used for an array equality.

Following that, the standard says, "terminating null character is always appended to the result", a nice way to express this property is using the logical *strlen* to compare the

length of the destination before and after the function call. Since, the correct value of the *strlen* guarantees that the string properly ends with a null character.

The last one would be trivial: a returning pointer is always equal to the original destination pointer (the **Returns** section of the function definition). Below are these three postconditions translated in ACSL:

```
ensures s_copied: array_equal{Post, Pre}(
        d, \old(strlen(d)),
        s, 0,
        \old(len(s, n))
    );
ensures sum: strlen(d) ≡ \old(strlen(d) + len(s, n));
ensures result_ptr: \result ≡ d;
```

2.2.4.2 Loop invariants

The first step in building loop invariants is to use *based* predicates to state that the bases for pointers `s` and `d` stay the same. Further, *based_ptr* is used to allow reasoning about memory accessed by `s`.

```
loop invariant based{Pre, Here}(&d);
loop invariant based{Pre, Here}(&s);
loop invariant s_mem: based_ptr{Pre, Here}(&s);
```

The next step is to define a "virtual" loop index. In this case, it can be done in multiple ways. So two invariants are defined to state that all these indexes are equal to each other:

```
loop invariant n_eq:
    \at(n, Pre) - n ≡ d - a - \at(strlen(d), Pre);
loop invariant s_eq:
    s - \at(s, Pre) ≡ d - a - \at(strlen(d), Pre);
```

Then, the index is limited between 0 and `len(s)`:

```
loop invariant limits:
    0 ≤ (d - a) - \at(strlen(d), Pre) ≤ \at(len(s, n), Pre);
```

Finally, the last invariant states that symbols from 0 to the "virtual" index were appended from source string to destination string:

```
loop invariant copied:
    ∀ ℤ j; 0 ≤ j < d - a - \at(strlen(d), Pre) ⇒
    \at(d, Pre)[j + \at(strlen(d), Pre)] ≡ \at(s, Pre)[j];
```

2.2.4.3 Triggering axioms

After adding all the loop invariants, Frama-C should be able to automatically prove all `strncat` postconditions except one about *strlen* of a new string (`ensures sum: strlen(d) ≡ \old(strlen(d) + len(s, n));`). Frama-C cannot prove this one because *strlen* is defined axiomatically, so it cannot just evaluate *strlen* to reason about its equality. It is only possible to reason about its value via defined axioms, which usually cannot be done automatically.

In order to help the automatic prover, one should provide asserts that will match the definition of the needed axiom with some variables substituted with correct values (16). In this particular case, the goal is to prove that the new *strlen* of `d` is equal to its old one plus `len(s)`. To achieve this, the `pos_or_nul` axiom is used with *forall* variable *i* replaced by `strlen(d) + len(s, n)` (split into two separate asserts):

```
/*@ assert a[\at(strlen(d) + len(s, n), Pre)] ≡ 0; */

/*@ assert ∀ ℤ j; 0 ≤ j < \at(strlen(d) + len(s, n), Pre) ⇒
                a[j] ≢ 0; */
```

2.2.4.4 Manual unfolding

Occasionally, the goal obligations could become very complex and the automatic prover might take a lot of time to prove certain goals or even not be able to prove them at all. This could be made worse by all added predicates in the goal specifications. In cases where a goal seems to have all information to be proven but nevertheless the automatic prover cannot verify, it might be helpful to manually unfold old custom predicate definition inside of the goal. It is done inside the "WP Goals" tab of Frama-C. Choose the needed goal on the list and select "Raw Obligation" or "Full Context" view mode, then select custom predicate and apply "Definition" tactic.

2.3 Conclusion

In this chapter, the process and steps required to verify string or array based functions were demonstrated. First, the procedure of converting the human language C standard definition of the function to formal requirements in the ACSL language were exemplified. Undoubtedly, the translation would always require individual treatment due to variances

in human specifications, nonetheless, presented examples could provide valuable directions for translation of similar functions.

Second, clear steps to construct loop invariants for cases that involve multiple pointer variables were proposed. To summarize, the steps are:

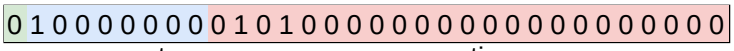
1. link all the pointer variables together using *based* predicates
2. define a "virtual" loop index derived from the relative positions of the pointers
3. set the lower and upper bounds for the "virtual" index
4. optional step is to provide a property expressing monotonicity of the defined index as evidence of loop termination
5. establish core invariants of the loop, these are invariants defining functionality of the loop, which could substantially vary between different cases

With usage of developed predicates and proven lemmas, these steps are arguably enough to verify many of the string library functions. Typically, with the appropriate definition of loop invariants, a function would be verified automatically (depending on the loop complexity or usage of other C language concepts, the last step might require extra lemmas or manual proof using external tools like Coq). This makes the verification process significantly more simple and accessible.

3

Building blocks for verification of floating-point functions

When facing a floating-point number in program verification, one has to deal with three different types to store the number and reason about it. The first one is the actual real number that floating-point is meant to represent. Usually, the real number cannot be stored in the memory precisely, so different rounding methods are used. The second type is the floating-point type itself, generally it is a single or double precision defined by the IEEE Standard for Floating-Point Arithmetic (IEEE 754). And the final type is a bit representation of the floating-point stored in the integer type.

$\mathbb{R}$	$x = 2.625_{10}$
IEEE 754	float x = 
bit representation	unsigned int i = 01000000001010000000000000000000

Conceptually a bit representation is no different from a floating-point number itself, both are stored as an identical array of bits. The only difference is the type used in a programming language to address it. Both integer and floating-point types provide different operations to perform on the number. Floating-point type provides general arithmetic and comparison operations, while integer type allows for more precise or efficient managing of the number. The integer bit representation is often used by C library implementations.

When working with Frama-C, all built-in operations including equality are performed on the real numbers. So, any lemma or condition defined in Frama-C has to be defined on real numbers and floating-point type has to be converted to real type. There is no way to reason about floating-point without casting it to real.

3.1 Unpacking double with logic functions

As mentioned earlier, the standard library implementations often need the integer bit representation of the floating-point number. The common way to obtain it is a "union trick". The difference between the floating-point number itself and its bit representation in integer is only the C language type used to access the memory. The trick allows to simply change the type of the memory from floating point to integer:

```
union {double f; uint64_t i;} ux = {x};
int ex = ux.i>>52 & 0x7ff;
int sx = ux.i>>63;
uint64_t uxi = ux.i;
```

However, Frama-C models all union fields as separate values, so this trick does not work. And it does not have any other tools to reason about floating-point bit representation. To work around that, all instances of the "union trick" are replaced by "abstract" C functions. These functions are defined without an implementation and only a Frama-C postcondition that states that the result of the function equals a particular part of the floating bit representation (sign, exponent or mantissa). Moreover, a `make_double` function is provided that performs an opposite trick by collecting all the bit representation parts back into the float-point number. Even though it requires a small modification of the original code, these functions can be used in place of a "union trick" in a way that Frama-C understands.

```
/*@ assigns \nothing;
    ensures x > 0 ⇒ \result ≡ 0;
    ensures x < 0 ⇒ \result ≡ 1; */
unsigned long long signbit(double x);

/*@ assigns \nothing;
    ensures \result ≡ exponent(x) + 0x3ff; */
unsigned long long exponent(double x);

/*@ assigns \nothing;
    ensures \result ≡ mantissa_64bit(x); */
unsigned long long mantissa(double x);

/*@ assigns \nothing;
    ensures \result ≡ make_double(sign, exponent, mantissa); */
double make_double(unsigned long long sign,
        signed long long exponent,
        unsigned long long mantissa);
```

Next, logical functions (`signbit`, `exponent` and `mantissa_64bit`) used in the postconditions need to be defined. Because Frama-C operates on the level of the real numbers, the functions also have to be defined on the real numbers. The result has to be a bit representation a real number would have if it would be assigned to a double precision floating-point. Additionally, functions here that start with a backslash (`\floor`, `\abs`, `\pow`) are the built-in math functions of Frama-C. All of them are defined as functions from $\mathbb{R}$ to $\mathbb{R}$.

```
    logic ℤ signbit(ℝ x) = x ≥ 0 ? 0 : 1;

    logic ℤ exponent(ℝ x) =
        \floor(\log(\abs(x)) / \log(2));

    logic ℤ mantissa_64bit(ℝ x) =
        \floor((\abs(x) / \pow(2, exponent(x)) - 1) * (1 << 52));
```

The simplest function out of the three is `signbit`, it returns the value of a first bit of a machine number (0 for positive and 1 for negative numbers). The next one is `exponent` that calculate the exponent of a real number. It is done by simply taking logarithm base 2

of the number. The `exponent` function does not consider the specific bit size of a resulting floating-point number, it is assumed that it will always fit and there is no overflow. What it means is that the exponent is never too large for a floating-point number to become `inf`. This assumption is based on the fact that even though the function is defined on real numbers, in reality it is always applied to a `double` floating-point.

The third one is `mantissa_64bit`, it returns a mantissa of a floating 64-bit number to represent a given real number. It first divides the number by 2^{exp}, normalizing it as if its exponent is 0, then it subtracts 1 because in the `IEEE 754` standard for floating-points the first 1 bit is implied for normal numbers. Lastly, it multiplies it by 2^{52} to convert it to an integer number. Important notice, if the original x number was already a `double` or `float` number, then the value of `mantissa_64bit(x)` must be an integer without `floor`.

Lastly, a function to convert back all three parts into a single real number is created. It basically performs reverse operations of the operations in unpacking functions:

```
logic ℝ make_double(ℤ s, ℤ e, ℤ m) =
    (m + (1 << 52)) /
    \pow(2, \floor(\log(m + (1 << 52)) / \log(2)) - e) *
    \pow(-1, s);
```

In order to link together bit representation and real number, all the logical functions have to be verified. One way of doing so is a lemma that states that performing unpacking and then packing back on a real number that is an exact `double` value will return the same value back. Verification of this lemma will be discussed later.

```
lemma make_double_from_parts:
    ∀ ℝ x; x ≡ (double) x ⇒ x ≢ 0 ⇒
        make_double(signbit(x), exponent(x), mantissa_64bit(x)) ≡ x;
```

3.1.1 fabs.c example

Using the unpacking mechanism above, it is now possible to verify a version of an absolute value function. To fully demonstrate a "union trick", this version features a full bit representation extraction, even though `fabs` only requires access to the sign-bit part of the value. It extracts all the bit representation parts into separate variables, assigns zero to the sign part and, finally, combines the value back together.

```
double fabs(double x)
{
    union {double f; uint64_t i;} u = {x};
    int sx = u.i>>63;
    int ex = u.i>>52 & 0x7ff;
    uint64_t uxi = u.i & (-1ULL >> 12);

    s = 0;

    u.i = (s << 63) & (ex << 52) & uxi;
    return u.f;
}
```

Now, the union has to be replaced with the functions created earlier so the Frama-C can reason about this function. In the modified version part variables `s`, `e` and `m` correspond to variables from the original: `sx`, `ex` and `uxi`.

```
double fabs(double x)
{
    signed long long s = signbit(x);
    signed long long e = signed_exponent(x);
    unsigned long long m = mantissa(x);

    return make_double(0, e, m);
}
```

With the "union trick" replaced, the function can be verified as any other C function. A contract for this function is very straightforward. From the ISO C standard: "The fabs functions return $|x|$." So, it ensures the result is equal to a logical absolute value function. Additionally, the standard does not specify any side effects or memory writings, so it assigns nothing. Finally, because the unpacking function above only accepts real numbers for the current version of a `fabs` function, a requirement for `x` to be finite is added. The full contract is below:

```
    requires finite_arg: \is_finite(x);
    assigns \nothing;
    ensures \abs(x) ≡ \result;
```

This contract cannot be verified automatically. A few lemmas are required to provide the WP plugin some assistance with it. An important property of the absolute value function is that it does not change any bit parts of the value except the signbit. This can be expressed in three separate lemmas about the value of logical unpacking functions. All three lemmas are proven automatically, no further assistance is required.

```
lemma abs_exponent: ∀ ℝ x; exponent(x) ≡ exponent(\abs(x));

lemma abs_mantissa:
    ∀ ℝ x; mantissa_64bit(x) ≡ mantissa_64bit(\abs(x));

lemma abs_signbit: ∀ ℝ x; signbit(\abs(x)) ≡ 0;
```

As noticed previously, all arithmetic in Frama-C is performed on the real number and not floating-point ones. This makes it important to observe that the absolute value of a floating-point variable can also be precisely stored in the same type of floating-point. Meaning that if the original input value of x is a `double` variable, the result value of $|x|$ can also be assigned to `double` without any information loss, i.e., rounding.

```
lemma abs_is_double: ∀ double x; \is_finite(x) ⇒
    \abs(x) ≡ (double) \abs(x);
```

This lemma is also verified automatically. With these four lemmas, WP can automatically verify the contract of a `fabs` function. This shows that in the case of the very basic floating-point function `fabs`, the proposed approach can be applied successfully without a need to verify any properties manually.

However, this example also immediately shows the limitation of this approach. The last lemma used in the verification defines a very critical property of the absolute value function. It is the function that performs exact computations on the floating-point value without rounding. This makes this function suitable for this approach. For functions where the result is not a value that can be represented exactly in a floating-point value, but has to be approximated by a floating-point number, verification becomes noticeably harder. In that case, one would also need to prove that the resulting approximation is indeed equal to a required real number rounded to a floating-point number. Many key floating-point functions in the C math library contain rounding in some form, nevertheless, this type of verification is out of scope for this work and left as potential future work.

3.2 The unpacking lemma

This section describes the specification and verification of the unpacking lemma. The goal of the lemma is to introduce equivalence between parts of the floating-point number and the original value. This lemma would allow to reason about values of the `signbit`, `exponent` and `mantissa` as actual parts of the double number and not as a separate functions. The core statement of the lemma should be:

```
make_double(signbit(x), exponent(x), mantissa_64bit(x)) ≡ x;
```

3.2.1 Specification of the lemma

However, as all logic statements in Frama-C are made in real number and not in floating-point numbers, this statement does not hold for any real number but only for real numbers that can be represented exactly by `double`. Two ways to represent this property have been tried.

The first attempted specification relies on defining `x` as a `double` value:

```
lemma make_double_from_parts:
    ∀ double x; \is_finite(x) ⇒ x ≢ 0 ⇒
        make_double(signbit(x), exponent(x), mantissa_64bit(x)) ≡ x;
```

However, this approach quickly proved to be unsuccessful — none of the automatic provers could use this lemma. The reason behind it is implicit type casts. All the folding and unfolding functions are defined on real numbers, thus making every use of `x` in the lemma cast to a real number. If one would make all the casts in the lemma explicit, it would contain four casts to $\mathbb{R}$:

```
lemma make_double_from_parts:
    ∀ double x; \is_finite(x) ⇒ x ≢ 0 ⇒
        make_double(
            signbit((ℝ) x),
            exponent((ℝ) x),
            mantissa_64bit((ℝ) x)) ≡ (ℝ) x;
```

To avoid multiple unnecessary casts, the `x` must be defined as a real value. Then another way of specifying that it is in fact `double` is required. It was found that an effective way of doing so is a condition that casting `x` to `double` will equal to `x` and not change its value (`x ≡ (double) x`). A full lemma defined this way can be seen below.

```
lemma make_double_from_parts:
    ∀ ℝ x; x ≡ (double) x ⇒ x ≢ 0 ⇒
        make_double(signbit(x), exponent(x), mantissa_64bit(x)) ≡ x;
```

3.2.2 An additional axiom

When generating proof obligations, the WP plugin does not expose the floating-point binary structure of a `double` type to the Why3 layer and automatic provers. It only generates a real-like type with lower and upper bounds. This makes it impossible to fully verify the `make_double` lemma without introducing any new axioms.

For the purpose of simplicity and to not build a full floating-point verification from the ground up, it was decided to create one high-level axiom such that this axiom should be enough to verify the rest of the lemma. It was chosen to use mantissa size of the `double` value for this axiom.

As defined earlier, `mantissa_64bit` part extraction function contains a `\floor` call in it that will drop all the bits of the mantissa past the 52nd bit. However, a `double` value contains exactly 52 bits of the mantissa (plus one implicit initial bit). Using this fact, an axiom was created stating that the value of `mantissa_64bit` equals its definition without the `\floor` call if `x` is `double`. The axiom uses the same way of defining `x` being `double` as in the earlier lemma specification.

```
axiomatic make_double {
    axiom mantissa_is_int:
        ∀ ℝ x; x ≡ (double) x ⇒ x ≢ 0 ⇒
            mantissa_64bit(x) + (1 << 52) ≡
                \abs(x) / \pow(2, exponent(x)) * (1 << 52);
}
```

3.2.3 Verification of the lemma

Verification of the unpacking lemmas was done in Coq. The full script can be found in the source materials provided on-line. To prove the lemma, two additional lemmas were required.

```
lemma make_double_help2:
    ∀ ℝ x; x ≢ 0 ⇒ 1 ≤ \abs(x) / \pow(2, exponent(x)) < 2;

lemma make_double_help:
    ∀ ℝ x; x ≡ (double) x ⇒ x ≢ 0 ⇒
        \floor(\log(mantissa_64bit(x) + (1 << 52)) / \log(2)) ≡ 52;
```

The first one states that any real number divided by its exponent would result in a value between 1 and 2. And the second one states that the exponent of the mantissa plus hidden bit is always 52. The proofs of both lemmas were done in Coq and can also be found in the sources.

3.2.4 Floor function

The WP plugin provides its own definition of a floor function when generating proof obligations for provers. However, this definition is very minimal and does not include any lemmas that would help to use this function in proofs. The Coq library has its own floor function called `R_Ifp.Int_part` and unlike WP it also includes a set of useful proven lemmas with it.

To use these lemmas on the WP version of floor, it is needed to provide a some kind of equivalence between two functions. To achieve it, a small Coq library was created containing the following lemma with a proof:

```
Lemma Floor_in_coq:
  forall (floor: R -> Z),
    (forall (x: R),
      (IZR (floor x) <= x)%R /\ (x < IZR (floor x + 1))%R) ->
  forall (r: R), floor r = R_Ifp.Int_part r.
Proof.
  intros floor Floor_down r.
  unfold R_Ifp.Int_part.
  apply Zplus_minus_eq.
  symmetry.
  apply R_Ifp.tech_up.
  rewrite Int.Comm.
  apply Floor_down.
  rewrite plus_IZR.
  rewrite Real.Comm.
  apply Rplus_le_compat_r.
  apply Floor_down.
Qed.
```

The lemma states that any function from $\mathbb{R}$ to $\mathbb{Z}$ that has the property $\forall x, (f(x) \leq x) \wedge (x < f(x+1))$ is equal to the Coq library function `R_Ifp.Int_part` and thus can be replaced by it.

This approach can be applied to many other mathematical functions for which the WP defining is lacking some crucial lemmas.

3.2.5 Delegation of non-linear arithmetic proofs to Frama-C

Occasionally, when working on a Coq proof non-linear real or bitwise expressions can be tricky especially for people less familiar with Coq. In that case, sometimes evaluating this arithmetic can be delegated to Frama-C. This will minimize the amount of work required to be done in Coq and allow keeping verification on the level of Frama-C as much as possible.

For example, during proof of the `make_double` lemma there was a need to proof the following arithmetic expression (here a $\ll$ sign mean the bitwise shift left operation):

$$\frac{\log(1 \ll 52)}{\log 2} = 52$$

This expression can be especially problematic to handle manually in Coq because it combines both non-linear real and bitwise operations. To avoid this, it is possible to create a simple lemma stating exactly the same expression. The lemma is proven automatically.

```
lemma real_2_ln_52:
    \log(1 << 52) / \log(2) ≡ 52;
```

This approach can also be used to provide some extra lemmas about bitwise operations to the Coq proof assistant. When generating proof obligations, Frama-C provides its own definition of bitwise operations with a bare minimum set of axioms. This sometimes makes proofs of bitwise operations in Coq slightly cumbersome.

For example, let's consider commutativity of bitwise and operations ($\forall x, y; x \wedge y = y \wedge x$). The WP internal simplifier QED is aware of the bitwise operations properties and can simplify them before exposing to external provers. So the lemma is defined normally as someone might expect it to be defined:

```
lemma l_land_commutativity_old:
    ∀ ℤ x, y; (x & y) ≡ (y & x);
```

The result generated after applying simplifications is just an axiom stating True (when generating the environment for the next lemma or condition, Frama-C defines all the previous lemmas as axioms abstracting away if the lemmas are actually proven or not):

```
Axiom Q_l_land_commutativity_old : True.
```

To work around that, the lemmas need to be intentionally made more complex, so the QED simplifier cannot reduce them to True. One way of achieving that is adding some extra variables that cannot be eliminated in the simplification process:

```
lemma l_land_commutativity:
    ∀ ℤ x, y, a, b; a ≡ y ⇒ b ≡ x ⇒ (x & y) ≡ (a & b);
```

The extra `a` and `b` variables cannot be evaluated by the simplifier because in general they can be any integer number. But the lemma only works if they are equal to `y` and `x` respectively. The extra variables do not limit the use of this lemma, as all automatic solvers can still apply it when the equality condition holds.

3.3 Bitwise operations

Many C math library implementations heavily rely on working with the bit representation of floating-numbers. This section shows some examples of verifying bitwise operations and discusses difficulties faced in the process.

3.3.1 Unsigned wrap-around example

Even though this example does not contain any floating-point operations, it is a basic function that requires similar techniques as verifying bitwise operations on bit representation of floating-point numbers. The function itself is just adding 256 to an unsigned char variable. According to the C standard, the result should be equal to an input value of the function:

```
/*@ ensures \result == x;
    assigns \nothing; */
unsigned char wrap_around (unsigned char x) {
    x = 256 + x;
    return x;
}
```

Despite the simplicity of the function, the postcondition cannot be verified automatically by automatic provers, so it has to be done manually in Coq. Before dealing with this function, let's consider the following lemma about casts to an unsigned char.

```
    lemma wrap_around:
        ∀ ℤ x; (unsigned char)x == (255 & x);
```

This lemma translated to Coq: `to_uint8 i = land 255 i` where `to_uint8` is a cast to an unsigned 8-bit integer. Because $255 = 2^8$ this can be easily rewritten as `to_uint8 i = to_uint 8 i` (notice the space in the second `to_uint`). Next the extraction axiom `is_uint8_ext` has to be used.

```
Axiom is_uint8_ext :
  forall (x:Numbers.BinNums.Z) (y:Numbers.BinNums.Z), is_uint8 x ->
  is_uint8 y ->
  (forall (i:Numbers.BinNums.Z), (0%Z <= i)%Z /\ (i < 8%Z)%Z ->
   bit_test x i <-> bit_test y i) ->
  (x = y).
```

The axiom means that if both `x` and `y` are 8-bit unsigned integer numbers and all the bits from 0 to 7 are equal, then `x` and `y` themselves are equal (where `bit_test x i` is a predicate that holds if the ith bit of a variable `x` is 1). To finish the proof, two other extraction axioms have to be used: `to_uint_extraction_inf` and `to_uint8_extraction_inf`. Basically, they both mean that casts to unsigned integer can be omitted when checking bits of lower rank (below 8 for `to_uint8` and below `n` for general `to_uint`).

```
Axiom to_uint_extraction_inf :
  forall (n:Numbers.BinNums.Z) (x:Numbers.BinNums.Z) (i:Numbers.BinNums.Z),
  (0%Z <= i)%Z /\ (i < n)%Z -> bit_test (to_uint n x) i <-> bit_test x i.
```

The proof of the original `wrap_around` function can be performed similarly but it only requires to test the 8th bit of the value `x`.

This shows that when working with bitwise operations, the integer number has to be treated not as a single variable but as an array of bits. It also explains why the function cannot be verified automatically. For theorem provers it is harder to reason about array types and especially in this case because the array is hidden behind the `bit_test` function. Furthermore, properties of arrays often require inductive reasoning to prove them, which is noticeably more difficult for automatic theorem provers and is not possible for solvers used in this work.

3.3.2 `remquo` example

The `remquo` computes the floating-point remainder of x/y. From the C standard:

> When $y \neq 0$, the remainder $r = x \; REM \; y$ is defined regardless of the rounding mode by the mathematical relation $r = x - ny$, where n is the integer nearest to the exact value of x/y; whenever $|n - x/y| = 1/2$, then n is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x. This definition is applicable for all implementations.

The `remquo` function mostly consists of bitwise operations, as it optimizes floating-point division on bit representations. This function was chosen as a more complex math function that showcases bitwise operations in the library.

Before verifying the correct functionality of the function, it makes sense to first try working only on proving absence of undefined behaviors and runtime errors. Often this can be achieved with significantly less effort and still provides a valuable verification result as it at least ensures error safety of the program.

First, for the sake of simplicity, the input values for the function are limited to normal finite real numbers. This allows to avoid some branching designed for the special cases of subnormal and infinite values.

```
    requires finite_arg_x: \is_finite(x);
    requires not_zero_x: x ≢ 0;
    requires not_subnormal_x: exponent(x) + 0x3ff ≢ 0;
```

Elimination of all the special cases allows to ignore multiple initial `if` statements of the function and focus on the main loop. The loop performs division x/y by subtracting y from x and reducing the exponent of the value x. Finally, in the second loop, it adjusts the exponent of the x if the first bit of it is not 1.

```
        for (; ex > ey; ex--) {                          /* 1st loop */
                i = uxi - uy.i;
                if (i >> 63 == 0) {
                        uxi = i;
                        q++;
                }
                uxi <<= 1;
                q <<= 1;
        }
        i = uxi - uy.i;
        if (i >> 63 == 0) {
                uxi = i;
                q++;
        }
        if (uxi == 0)
                ex = -60;
        else
                for (; uxi>>52 == 0; uxi <<= 1, ex--);  /* 2nd loop */
```

For the first loop it is trivial to verify absence of runtime-errors by setting limits on possible values of `ex`. It strictly decreases and is always greater than `ey`. These properties are enough for the plugin to deduce safe behavior of the loop.

```
    /*@ loop invariant \at(ex, LoopEntry) ≥ ex ≥ ey;
        loop assigns ex, i, uxi, q;
        loop variant ex - ey; */
```

However, the second loop is not that trivial despite its length. What it does is shifting bits of the `uxi` to the left until bit 52 is 1. Similar to the previous loop, some limits on the number of loop iteration are required. Because the value of `uxi` cannot be 0 (it is handled by the `if` case before the loop) it is guaranteed that mantissa `uxi` contains at least one non-zero bit. During the left-shifting process, the bit will eventually become the 52nd and the loop will terminate. The one difficulty is that a non-zero value containing a non-zero

bit is not a trivial property for Frama-C, so it has to be verified separately. The property can be written as `assert` as follows:

```
assert exists: ∃ ℤ i; 0 ≤ i ≤ 52 ∧ ((uint64_t)(uxi << i))>>52 ≢ 0;
```

Attempting to verify this `assert` shows limits of working with the bit representation of the integer variable. The bit representation, as mentioned before, is much more of an array type than a number type. But it is hidden inside an integer type for the purpose of efficiency, and one of the most significant limitations is the inability to access the value of a single bit. The only way of doing so is performing some combinations of shift and bitwise-and operations. This property was not verified in this work.

Now, as with all loops, there is a need for a limit on the index of the loop. Even though this loop does not have an implicit index, the value of `ex` can serve as an index because it is decremented in every loop iteration. The loop will end when the 52nd bit finally becomes 1, so existence of a non-zero bit in the `uxi` can be used to set a limit on the number of loop iterations. The loop invariant based on this can be seen below, it is proven automatically by Frama-C.

```
loop invariant ∀ ℤ i; 0 ≤ i ≤ 52 ⇒
    ((uint64_t)(\at(uxi, LoopEntry) << i))>>52 ≢ 0 ⇒
        \at(ex, LoopEntry) - ex ≤ i;
```

The final step to verify the loop is to observe how the value of `uxi` is changed by the loop. Each iteration loop performs a single left shift on the `uxi`, so on each iteration the `uxi` is equal to its original value shifted by the value of index (which is current `ex` minus original `ex`).

```
loop invariant uxi ≡
    (uint64_t)(\at(uxi, LoopEntry) << (\at(ex, LoopEntry) - ex));
```

The invariant can be proven automatically with the help of the two following lemmas. The second lemma is also proven automatically from the first one. The first one has to be proven manually. The `lsl_touint64` lemma states that the second cast to an unsigned 64-bit integer can be removed when performing a left shift. The proof of the lemma is similar to the wrap-around example and relies on applying several extraction axioms.

```
lemma lsl_touint64:
    ∀ ℤ x, y; 0 ≤ y ⇒
        (uint64_t)(((uint64_t)x) << y) ≡ (uint64_t)(x << y);

lemma lsl_touint64_lsl:
    ∀ ℤ x, y, z; 0 ≤ y ⇒ 0 ≤ z ⇒
        (uint64_t)(((uint64_t)(x << y)) << z) ≡
            (uint64_t)(x << (y + z));
```

With all the above, Frama-C can finally automatically verify that the `remquo` function is safe from undefined behaviors and runtime errors. However, even this simple step showed some difficulties. It displays that even without floating-point to real value conversions and other floating-point specifics, the math functions rely heavily on bitwise operations, which often lead to conditions that cannot be verified automatically. Most of them can still be proven manually in Coq, but it requires too much extra resources to be efficient for industry environments.

3.4 Conclusion

In this chapter, a method to link floating-point variables to their bit representation has been discussed. The common way of extracting bit representation via `union` used by library implementations cannot be used directly in verification. This union behavior is not defined by the C standard, but is supported by all main-stream C compilers.

As an alternative, the extraction part with union can be replaced by abstract functions which are specified to return parts of bit representations using function contracts. This approach can be applied to some simple math functions, here the `fabs` and `remquo` functions are used as an example.

From the demonstrated examples it can be observed that current tools are generally not proficient enough to keep the verification of floating-point functions on the level of Frama-C toolkit as it is with string functions. It applies to both work with floating-point variables themselves and with their bit representations. The inevitable requirement to delve deeper to the level of manual proof, despite the help from proof assistant, makes floating-point verification distinctly less accessible than string and array based verification. To address these difficulties, a few techniques have been suggested that could help automate or simplify at least some parts of the process.

4

Conclusion

This thesis showed a few approaches to take when verifying a C library function. The main focus of the work is not only the verification of different functions from the libraries, but also making the process more accessible. If the formal verification cannot be performed without in-depth knowledge of the formal methods or immense amounts of extra resources, then in most practical cases an alternative verification method will be used. The alternatives include static code analysis, data and control flow analysis, and many others.

The thesis has been done in collaboration with Solid Sands. The company provides technology for C and C++ compiler and library testing. Even though this work is not directly connected to the current product of the company, it is a research project of formal verification as alternative to other testing and qualification techniques. Since the research is aimed at accessibility, it could help future development of formal method tools designed for C programmers.

When dealing with a pointer oriented functions, such as library functions for string manipulation, it was shown that formal verification can be a competitive alternative to other methods. The most part of the process can be performed with only usage of the Frama-C toolkit with the help of axioms and predicates introduced in this work. This massively simplifies the formal verification process and makes it available for industrial use.

A method to verify similar string functions with concrete and easy to replicate steps was suggested. A special focus was given to loops featuring multiple pointer variables without an explicit index variable. This is a very common pattern in C library implementations and in general C code. With the help of provided lemmas and predicates, it can be effortlessly verified completely automatically, relaying only on ACSL annotations.

However, when working with floating-point functions, the current tools are not sufficient to keep everything automatic and on the level of Frama-C specifications. These functions would often require manual proof of most of the conditions. Moreover, a common usage of unspecified `union` behavior in the library implementations requires unavoidable modifications to the source code. As one such modification, this work shows the use of "abstract" functions, i.e., functions without bodies that only have contracts. A set of such functions was developed to calculate bit representations of real numbers. The set is not fully verified and relies on one unverified lemma (and as such is defined as an axiom) but can be used to work with floating-point functions as a replacement for the `union` based approach.

In addition to obstacles caused by floating-point variables themselves, the work with bitwise operations turned out to be surprisingly more challenging than expected. One of the directions for future work is to make bitwise operations more accessible for automatic verification with Frama-C. The possible way of doing so is building a framework to treat bit-representations as a proper array type.

In this thesis, only a small fraction of the C library was considered. The library is not limited by pointer and floating-point operations. There are multiple other sections that could be looked into in the future. These sections include IO functions or functions with external state (for example wide-character functions). These might require some model of the external environment, like the underlying file system. Nevertheless, from the current experience with the Frama-C toolkit, it displays potential to be able to handle them mostly automatically.

Appendix A

Appendix

A.1 Proof of the first null symbol lemma

In the proof, a native support for Coq of the WP plugin is used. Despite the fact that it is marked as deprecated native support, it allows for easier access to some C specific axioms and lemmas in Coq.

To perform an induction proof of the `exists_first` lemma, a stricter version of the lemma is used which states that for all i either there are no null symbols before i in string `s` or there exists a first null symbol before i.

```
lemma exists_first_2:
    ∀ char* s; ∀ ℤ i;
        0 ≤ i ⇒ (
            (∃ ℤ i1; (0 ≤ i1 ≤ i ∧
                (∀ ℤ j; 0 ≤ j < i1 ⇒ s[j] ≢ '\0') ∧
                s[i1] ≡ '\0')) ∨
            (∀ ℤ j; 0 ≤ j ≤ i ⇒ s[j] ≢ '\0'));
```

The stricter version of the lemma simplifies inductive reasoning as now the inductive step is very straight forward. On the $i + 1$ step there are two cases. The first case, there were no null symbols before, meaning everything depends on the $(i+1)$th symbol as either it is the first null or there are no nulls at all. The second case, there was a first null symbol before i meaning it is the same first null symbol before $i + 1$. The full Coq script with some comments of the proof can be seen below.

```
Goal.
  forall (i : Z),
  forall (t : farray addr Z),
  forall (a : addr),
  ((0 <= i)%Z) ->
  ((is_sint8_chunk t)) ->
  ((forall (i_1 : Z), ((i_1 <= i)%Z) -> ((0 <= i_1)%Z) ->
    (((t.[ (shift_sint8 a i_1%Z) ]) <> 0)%Z)) \/
   (exists i_1 : Z, (((t.[ (shift_sint8 a i_1%Z) ]) = 0)%Z) /\
    ((i_1 <= i)%Z) /\ ((0 <= i_1)%Z) /\
    (forall (i_2 : Z), ((0 <= i_2)%Z) -> ((i_2 < i_1)%Z) ->
     (((t.[ (shift_sint8 a i_2%Z) ]) <> 0)%Z)))).
Proof.
  intros.
  induction i using Z_induction with (m := 0%Z).

  (* Base of the induction *)
  + assert (i = 0); auto with zarith.
    assert (
      (t .[ shift_sint8 a 0] = 0) \/ (t .[ shift_sint8 a 0] <> 0)
    ) as choice.  tauto.
    destruct choice.

    (* First symbol is 0 *)
    * right.
      exists 0.
      split; auto with zarith.

    (* First symbol is not 0 *)
    * left.
      intros.
      rewrite H2 in *.
      assert (i_1 = 0); auto with zarith.
      rewrite H6 in *.
      auto with zarith.
```

```
  (* Step of the induction *)
  + destruct IHi; auto with zarith.

    (* Case with no 0 before *)
    * assert (
        (t .[ shift_sint8 a (i + 1)] = 0)
        \/ (t .[ shift_sint8 a (i + 1)] <> 0)
      ) as choice.  tauto.
      destruct choice.

      (* New symbol is 0 *)
      - right.
        exists (i + 1).
        split; auto with zarith.

      (* New symbol is not 0 *)
      - left.
        intros.
        assert ((i_1 = (i + 1)) \/ (i_1 < (i + 1))) as cc. {
          apply or_comm.
          apply Zle_lt_or_eq.
          auto.
        }
        destruct cc.
        -- rewrite H6 in *.
           auto with zarith.
        -- assert (i_1 <= i).
           auto with zarith.
           apply H2; auto with zarith.

    (* Case where there is 0 before *)
    * right.
      destruct H2.
      destruct H2.
      destruct H3.
      destruct H4.

      exists x.
      split; auto with zarith.
Qed.
```

A.2 Unsafe casts in the `memset` function

Due to technical limitations the Frama-C treats a void pointer as a signed 8-bit integer pointer. This could sometimes lead to unsafe casts. One example of it is a `memset` function. The standard definition for it is the following:

> `void *memset(void *s, int c, size_t n);`
>
> The memset function copies the value of c (converted to an `unsigned char`) into each of the first n characters of the object pointed to by s.

This explicitly defines a cast from an `unsigned char` value to a `void` value. From the Frama-C perspective it is a cast between a signed 8-bit integer and unsigned 8-bit integer.

There are two approaches to verify a function in such a case. It is possible to use the WP typed memory model with unlimited casts. However, this is an unsound model, and it might require extra caution. The second approach is to change the function to work with `signed char` instead of the `unsigned` one and avoid unsafe casts completely. Below is a description showcasing the second approach.

A.2.1 Function contract

```
/*@
    requires valid_dest: \valid((char *)dest + (0 .. n - 1));
    assigns dest: ((char *)dest)[0 .. (n - 1)];
    assigns \result \from dest;

    ensures set: ∀ ℤ i; 0 ≤ i < n ⇒
        ((char *)dest)[i] ≡ (char)c;
*/
void *memset(void *dest, int c, size_t n);
```

References

[1] Robert W Floyd. **Assigning meanings to programs**. In *Program Verification*, pages 65–81. Springer, 1993.

[2] Charles Antony Richard Hoare. **An axiomatic basis for computer programming**. *Communications of the ACM*, **12**(10):576–580, 1969.

[3] Edsger W Dijkstra. **Guarded commands, nondeterminacy and formal derivation of programs**. *Communications of the ACM*, **18**(8):453–457, 1975.

[4] The Coq Development Team. **The Coq Reference Manual**. Available at `https://coq.inria.fr/doc/`.

[5] Makarius Wenzel, Lawrence C Paulson, and Tobias Nipkow. **The isabelle framework**. In *International Conference on Theorem Proving in Higher Order Logics*, pages 33–38. Springer, 2008.

[6] Konrad Slind and Michael Norrish. **A brief overview of HOL4**. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

[7] Michael Norrish. **Deterministic expressions in C**. In *European Symposium on Programming*, pages 147–161. Springer, 1999.

[8] Xavier Leroy. **Formal certification of a compiler back-end or: programming a compiler with a proof assistant**. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54, 2006.

[9] Robbert Jan Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud Universiteit Nijmegen, 2015.

[10] Sascha Böhme, Michał Moskal, Wolfram Schulte, and Burkhart Wolff. **HOL-Boogie—An interactive prover-backend for the Verifying C Compiler**. *Journal of Automated Reasoning*, **44**(1-2):111, 2010.

[11] Thomas Ball, Brian Hackett, Shuvendu K Lahiri, Shaz Qadeer, and Julien Vanegue. **Towards scalable modular checking of user-defined properties**. In *International Conference on Verified Software: Theories, Tools, and Experiments*, pages 1–24. Springer, 2010.

[12] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. **Frama-C: A software analysis perspective**. *Formal Aspects of Computing*, **27**(3):573–609, 2015.

[13] Oleg Mürk, Daniel Larsson, and Reiner Hähnle. **KeY-C: A tool for verification of C programs**. In *International Conference on Automated Deduction*, pages 385–390. Springer, 2007.

[14] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. **VCC: A practical system for verifying concurrent C**. In *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[15] Bart Jacobs and Frank Piessens. **The VeriFast program verifier**. Technical report, Citeseer, Katholieke Universiteit Leuven, 2008.

[16] Loïc Correnson Zaynah Dargaye Allan Blanchard Patrick Baudin, François Bobot. **WP Plug-in Manual, Frama-C 23.1 (Vanadium)**. Available at `https://frama-c.com/html/documentation.html`, CEA-List, Université Paris-Saclay.

[17] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. **Alt-Ergo 2.2**. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, 2018.

[18] Leonardo De Moura and Nikolaj Bjørner. **Z3: An efficient SMT solver**. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[19] Dillon Pariente and Emmanuel Ledinot. **Formal verification of industrial C code using Frama-C: a case study**. *Formal Verification of Object-Oriented Software*, page 205, 2010.

[20] Jinhu Li and Jeffrey S Racine. **Maxima: An open source computer algebra system**. *Journal of Applied Econometrics*, **23**(4):515–523, 2008.

[21] **Maxima Documentation**. Available at `https://maxima.sourceforge.io/documentation.html`.

[22] David Monniaux. **The pitfalls of verifying floating-point computations**. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **30**(3):1–41, 2008.

[23] Geoff Barrett. **Formal methods applied to a floating-point number system**. *IEEE transactions on software engineering*, **15**(5):611–621, 1989.

[24] Sylvie Boldo and Jean-Christophe Filliâtre. **Formal verification of floating-point programs**. In *18th IEEE Symposium on Computer Arithmetic (ARITH'07)*, pages 187–194. IEEE, 2007.

[25] Jean-Christophe Filliâtre and Claude Marché. **Multi-prover verification of C programs**. In *International Conference on Formal Engineering Methods*, pages 15–29. Springer, 2004.