Scania CV AB

EPXS Systems Safety Architecture

Summer Research Project Report

# First Steps towards Floating-Point Verification in an Industrial Setting with `Frama-C`

*Forrest Timour*

supervised by
Christian Lidström

September 19, 2018

# Contents

**Abstract**

One of the most well-known sources of bugs in industrial code is inappropriate handling of floating point arithmetic. Scania has an experimental framework for formally verifying specifications through code annotations, but it lacks any support for floating-point numbers. My goal was to look into the viability of the open-source tool `Frama-C` for back-end verification, and to see what kinds of reasoning we can do in our specifications to verify safety properties involving real numbers and their corresponding floating-point representations. This report focuses on a heavily-used library function for interpolation and the progress that was made towards automatically verifying its correctness. The challenges and possible pitfalls along the way will be explained in detail, as well as possible next steps for the future of safe and verified embedded C code at Scania.

# 1 Introduction

Floating-point arithmetic is used often in many settings, but is one of the most difficult things to reason about soundly in a programming scenario. Errors aren't always obvious, and they don't always show up right away. One of the most famous examples of this is the US patriot missile failure of 1991 [2], where a small rounding error, compounded over 100 hours, became significant enough to cause the deaths of 28 people. In that case, the error was only discovered as a result of this catastrophic failure. Awareness of the problem has increased since then, but the issues have not been completely solved. This is incredibly important to us at Scania, especially as we move forward into the age of fully autonomous vehicles. We can create safety specifications to try to account for all kinds of life-threatening situations that might arise, but at the end of the day we need a way of being sure that those specifications are followed and that no action is unexpected or dangerous.

There has been a push for automated safety and specification verification research at Scania, especially as autonomy increases and regulations become stricter. The most safety-critical onboard embedded systems are being analyzed [1], in order to make sure that control flow is correct, memory accesses are valid, and no unexpected behaviour can occur. Now, as we extend this to include analyzing floating-point behaviour, our focus will be on overflow, underflow, rounding errors, and the differences between real and computer arithmetic. With the work we are doing now and in the future, we hope to achieve a level of assurance that simply was not possible before.

# 2 Floating-Point Arithmetic

## 2.1 Basic Information

Computers can handle integer calculations in binary with single machine instructions and with intuitive behaviour, as long as overflow is handled properly. When we move into the realm of representing real numbers, the problem becomes that the number of significant digits we need to keep track of can explode very quickly. It's not feasible to keep track of all of them, so we must approximate in some way. With fixed-point numbers, we choose a precision we want to be able to keep track of, but the choice of precision limits the magnitude of numbers we can represent because of the limited number of bits we have available. Floating-point numbers are the solution to this magnitude problem, but bring with them a host of other issues that have to be dealt with.

A floating-point number can be thought of as a triple of three separate values – a sign $s$ (+/-), an exponent $e$, and a significand (or mantissa) $m$ of precision $p$. There is also an implicit parameter $\beta$ that is the base, which we will assume to always be 2.

Using these three numbers with appropriate encodings, we can approximate $\mathbb{R}$ with $\mathbb{F}$, the numbers $x$ encoded by $x = (-1)^s * m \times \beta^e$. The number of significant digits is still limited to $p$, but they can be of any magnitude encodable by the exponent.

We have a few basic theorems [5] that have been proven about floating-point numbers, for example that relative error of subtraction can be as large as $\beta - 1$. A relative error of 1 is not so helpful, so we can improve that using extra guard digits for subtraction to a rounding error of less than $2\epsilon$, where $\epsilon$ is one half the value of the least significant digit, given the resulting $e$. There is still a lack of portability and usability with such a basic view of floating-point numbers, so eventually a standard emerged in 1985. We still use this standard today, and it will be the basis for the reasoning we are able to do.

## 2.2 IEEE-754 Specification

The IEEE-754 specification defines arithmetic formats, encodings, rounding rules, operations, and exceptions for floating point numbers. We will focus specifically on the `binary32` format, but all of the reasoning we do can easily be extended to larger or smaller precisions.

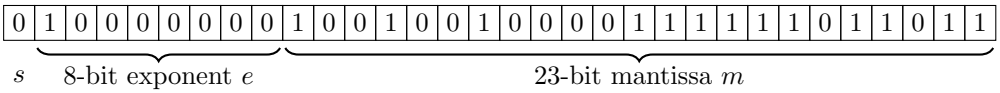



Figure 1: bit diagram of a 32-bit float representing $\pi$

Figure 1 shows an example of a floating-point bit vector encoding. The $s$ bit is either 0(+) or 1(-). If we were to treat $e$ as a signed integer, its range would be $[-128, 127]$, or $[1000\ 0000, 0111\ 1111]_b$. These binary values aren't uniform bit vectors however at the hardware level, so we instead represent the *biased* exponent $e_b$ as an unsigned integer in the range [0,255]. Extracting the "real" exponent $e$ is done by subtracting the *bias*, 127.

The mantissa can be interpreted in two ways: normal or denormal. Normalized floating-point numbers have an implicit leading 1 bit, making the effective precision 24 bits instead of 23. However, this makes it impossible to encode anything below $\beta^{e_{\min}}$, called the "underflow gap". Denormal numbers do *not* have this implicit leading 1, and they are used to fill this gap. Denormals are restricted to the exponent value $e_{\min}$, so we can visualize the encodings as seen in Figure 2.

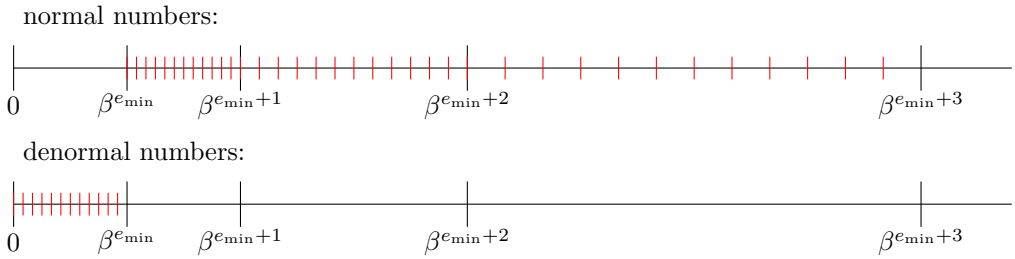



Figure 2: distribution of positive floating-point normal and denormal numbers. Negative numbers are exactly symmetric, including -0.

Notice that the distribution of normalized values is not uniform, as there are $2^p$ encodable values in the range of each exponent $[\beta^{e'}, \beta^{e'+1})$. The size of this range increases by a factor of $\beta$ as $e$ increases. As such, we can approximate small numbers

quite well with 24 bits of precision, but large numbers with the same bit precision will be less "precise" in their simulation of $\mathbb{R}$. The distance between adjacent denormal numbers is the same as the distance of those in $[\beta^{e_{\min}}, \beta^{e_{\min}+1})$.

That takes care of numbers near zero, but there must be an encoding for denormals and the "special" values $\pm$`Inf` and `NaN`. In the biased exponent values 0-255, we reserve the value 0 for denormal numbers with $e = -126$, we reserve the value 255 for $\pm$`inf` and `NaN`, and the remaining values 1-254 are for normalized numbers with $e \in [-126, 127]$. This is all summarized in Table 1.

| biased $e_b$ | mantissa | interpretation |
|---|---|---|
| 0 | — | denormal number with $e = -126$ |
| 1-254 | — | normal number with $e = e_b - 127$ |
| 255 | $= 0$ | $\pm$`inf` (depending on $s$) |
| | $\neq 0$ | `NaN` |

Table 1: biased exponent interpretation

Next, we have rounding modes. There are five different rounding modes described in the standard:

| **Rounding Modes** |
|---|
| round towards $\infty$ |
| round towards $-\infty$ |
| round towards 0 |
| round to nearest, ties away from 0 |
| round to nearest, ties to even |

The first two are absolute directions, the third is simple truncation, and the last two are ways of conventional rounding, to the closest value. The default is "round to nearest, ties to even", but this can be changed within a program.

Finally, the standard mandates *exact rounding* of basic operations $[+, -, \times, /]$. The rounded versions of these operations $[\oplus, \ominus, \otimes, \oslash]$ are guaranteed to have the same result as if the operation were carried out with infinite precision and then rounded. The advantage of this is that we can easily put bounds on the error of operations using just the precision and exponent of the result.

## 2.3 Challenges and Pitfalls

There are many challenges that arise from the complicated nature of the IEEE-754 floating-point representation, and we can fit them into six categories:

### 2.3.1 Overflow

A floating-point overflow happens when the result of a calculation is greater than the maximum normal floating point numbers, or less than its additive inverse. In these cases, the exponent can't be used to maintain the normalized format, so the result is set to $\pm\infty$. This is similar to integer overflow, except that we have a special value instead of simply wrapping around to a clearly incorrect result.

Consider, for example, the average function in Figure 3. While mathematically correct, the temporary variable `sum` might be too large to be represented if `a` and `b` are large enough in magnitude and have the same sign. In this case, `sum` becomes `+inf`, and `+inf` is returned, even though the *true* average *can* be rounded to a floating-point number.

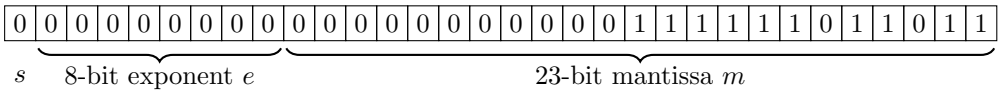
```
float avg(float a, float b) {
    float sum = a + b;
    return sum / 2.f;
}
```

Figure 3: an incorrect average function

### 2.3.2 Underflow

In some older systems not using the IEEE-754 standard, there was only one usable number in the "underflow gap", namely 0. If the result of any operation was less than $\beta^{e_{\min}}$, it would be flushed to zero. In the IEEE-754 model with denormal numbers and rounding to nearest, we get gradual underflow instead.

When talking about denormal numbers, there are two ways to imagine the bit-representation. The first is 23-bit precision with the possibility of any number of leading zeroes. In this case, there are $2^{23}$ denormal numbers with $e = -126$. The other view is that the leading zeroes become included in $e$, and the precision is reduced by one or more bits. Now, there are $2^{22}$ numbers with 23-bit precision and $e = -127$, $2^{21}$ with 22-bit precision and $e = -128$, $\ldots 2^{i-1}$ numbers with $i$-bit precision and exponent $i - 150$, ... and 0. Consider the following example:




This number can either be interpreted as $0.00000000000111111011011_b \times 2^{-126}$, in the format described in the standard, or equivalently as $1.11111011011_b \times 2^{-138}$. In this second case, the effective precision (from the first 1 to the last significant digit) is only 12 bits. By gradual underflow, we mean that as numbers in the underflow gap get closer to zero, they lose more and more precision. In the 23-bit precision view of denormal floating-point numbers, this means that the rounding error is *more significant* relative to the magnitude of the result.

### 2.3.3 Rounding Error/Non-Representable Numbers

One of the consequences of using a binary system, when we usually use base 10 in our daily lives, is that many simple and well-understood numbers are not even representable, and have to be approximated before applying any operations to them. We expect that some mathematical constants (for example $\pi$) will be approximated, so we might build in tolerances for this. Other simple constants, like $\frac{1}{10}$, are infinitely repeating fractions in binary. The rounding of this very value was the cause of the missile failure mentioned in the introduction.

When rounding or representation error compounds over many operations, it can be difficult to find and diagnose. The maximum representable floating point number is

$$\texttt{FLT\_MAX} = 340282346638528859811704183484516925440.000000$$

Let's say we want to count from the number 1 to the number 100,000,000. Consider the code snippet in Figure 4. The constant 100000000 is much less than `FLT_MAX`, and it *is* representable as a floating-point number. The problem is that `f` is rounded at every loop iteration, and the consistent rounding down leads to an infinite loop. Specifically,

$$(\texttt{float})(\texttt{16777216.f} + 1) = \texttt{1677216.f}$$

```
for (float f = 0; f < 100000000; f++) {
    printf("%d\n",f);
}
```

Figure 4: rounding error resulting in an infinite loop

In this case, and in many other similar situations, small rounding errors can compound over time to cause large problems.

#### 2.3.4 IEEE-754 non-numbers

We already mentioned overflow, when a result becomes `+/- inf`. Infinities can also be produced accidentally with division by zero, or by values near zero. As soon as one value goes to infinity, it is often the case that results are no longer useful, or that the infinities propagate or become `NaN`. This can often be a result of unexpected underflow or overflow. If `NaN` is input to any basic operation, the result will also be `NaN`. It has unusual properties, like being the only number for which `x == x` evaluates to `false`.

#### 2.3.5 Hardware Support

The next pitfall isn't a result of the IEEE-754 specification itself, but rather how it is followed. There has been a push for standardization, but not all CPUs have integrated hardware support for IEEE-754. Some (especially older) CPUs use Intel's x87 FPU (floating-point unit), which is infamous for internal 80-bit precision that leads to unexpected and inconsistent results [8].

In cases where there is no hardware support for the standard, it is sometimes still possible to emulate the correct behaviour using software. This leads to performance hits though, and can be a problem in embedded systems where performance is critical.

#### 2.3.6 Compiler/Thread Settings

Finally, the compiler itself can be the source of many floating-point issues. As mentioned before, denormal numbers can sometimes be turned off to improve performance. Even if code is proven correct and bug-free, it can be compiled to incorrect binaries. Compiling with different floating-point modes, partial implementations of the standard, different levels of optimization, using different hardware, and even bugs in the compiler itself can have unexpected consequences and be the source of problems.

One of the most difficult-to-diagnose problems can be rounding-mode errors. In some implementations, the rounding mode is thread-local, and there have been cases where library calls or device drivers can change the rounding mode and not change it back [10]. This all has to be taken into account when verifying the safety of embedded systems, in addition to proving the correctness of the code itself.

# 3 Linear Interpolation Case Study

## 3.1 C Programming at Scania

At Scania, there are strict guidelines for what is allowed in the C programs for embedded systems. Unfortunately, even the internationally recognized ISO/IEC standards are under-specified in places. It is not clear, for example, at what precision intermediate floating-point values are stored and evaluated, and when the rounding occurs. According to the ISO/IEC-9899 C language specification, "The values of floating operands and

of the results of floating expressions may be represented in greater precision and range than that required by the type". It comes down to the compiler settings used to generate the assembly code to decide the precision that will be used and where rounding will be applied in complex floating-point calculations!

```
[Sensitive details have been removed]
```

### 3.2 Util_Ipol

We analyzed a core library function that performs linear interpolation over floating-point values. The actual floating-point error that we're interested in analyzing comes from the abstract function `ipol` (Figure 5):

```
// precondition: 0.f <= t <= 1.f
//               a <= b
//               b - a <= int16_t_MAX
float ipol(float a, float b, float t) {
    float t_inv  = 1-t;
    float left   = a * t_inv;
    float right = b * t;
    return left + right;
}
```

Figure 5: Simplified interpolation function. The full version has been omitted.

Here everything is stored without intermediate values, so we can be explicit in our analysis of where rounding errors can occur. This must be, as mentioned before, verified through the compiler settings used because of the ambiguity in the C language specification. In order to proceed with the verification, we need to define what *correct behaviour* is for the function. The first step was to create a specification, which has been divided into six parts below, heavily inspired by a previous specification for an average function [3].

| **Property** | Description |
|---|---|
| **Endpoint Interpolation** | `ipol(a,b,0.f) = a` and `ipol(a,b,1.f) = b` |
| **Bounded Interpolation** | `t` $\in [0,1] \rightarrow$ `result` $\in [a,b]$ |
| **Order Invariance** | `ipol(a,b,t) = ipol(b,a,(float)(1-t))` |
| **Bounded Error** | Interpolation is correct within a few *ulp*'s |
| **Absence of Overflows** | Overflow does not occur |
| **Sign invariance** | `ipol(a,b,t) = -ipol(-a,-b,t)` |

In interpolation, $t = 0$ and $t = 1$ are special cases where we expect no rounding error. **Endpoint Interpolation** is a formalization of this to independent arguments, to show that for example `ipol(a,b,0)` returns $a$, independent of the value of $b$. **Bounded Interpolation** is a basic monotonicity property of the interpolation formula with real numbers, and we want to verify that it also holds for floating-point numbers. **Order Invariance** and **Sign Invariance** establish the fact that the interpolation result is not a product of the order or sign of parameters, but rather the underlying numbers. Finally, **Absence of Overflows** is a condition for correctness and **Bounded Error** is an attempt to prove the best bound that we can in order to help prove safety properties of the greater system. In our specific case, we make the assumption that `a <= b`, so we will combine **Order** and **Sign Invariance** into one property to maintain this precondition.

# 4 Formal Systems for Verification

## 4.1 Coq and Human-Assisted Proofs

In order to do the verification, there has to be some trusted, automated framework that can check proofs for errors and verify when they are correct. Coq is one such formal proof management system, that supports both automated proof solving attempts and human interaction. It provides a formal language to describe algorithms and properties, for which the only valid syntax is that which has been proven. This language is a functional type language with lambda calculus at its core.

We will show a brief example here, but the reader is referred to other sources to gain a full understanding of the language and syntax [11]. Consider, for example, a formal proof of the tautology $A \to (A \to B) \to B$. This can be proven in Coq with the script in Figure 6.

```
Theorem thm1: forall A B : Prop, A -> (A -> B) -> B.
Proof.
    intros A B.
    intros proof_A.
    intros A_imp_B.
    pose (proof_B := A_imp_B proof_A).
    exact proof_B.
Qed.
```

Figure 6: a basic Coq script proving a tautology of implications

In the proof, Theorem `thm1` is stated in terms of *properties* (`Prop`), which are the basic units of Coq proofs. The properties are general so that they can be applied to any abstract object. In fact, an instantiation of an "object" in Coq is typically just a type containing some data along with a *proof* that certain properties hold.

The script part of the proof is between `Proof.` and `Qed.`, and is made up of predefined tactics using lambda calculus syntax. The tactic `intros` defines variables and makes assumptions for direct implication proofs. In the example above, `A` and `B` are instantiated automatically as arbitrary properties, and then `proof_A` and `A_imp_B` are assumed as proofs of type `A` and `A -> B` respectively. The tactic `pose` creates a new proof of type `B`, `proof_B`, by applying `proof_A` as an argument to `A_imp_B`. Finally, `exact` discharges the proof goal because the consequent has been realized.

The whole proof can be written interactively in the Coq IDE. Starting from an empty script, it is possible to try tactics and prove additional theorems in order to reach a final proof goal. The current goals and available hypotheses are always available in one side, and the proof can be completed in an interactive, problem-solving manner. Every step can be type-checked and verified along the way, and then the proof script can be saved and automatically verified later. If we can create a proof script that helps prove our safety properties, then it can be parsed again later to automatically re-verify whatever properties we need!

## 4.2 Floating Point Formalization Methods

There have been multiple attempts at formalization and axiomatization of floating-point arithmetic, and each method of representation has its own advantages and disadvantages. We will first look at a general overview of how this formalization works and is useful, then look more in depth at a specific library developed for Coq.

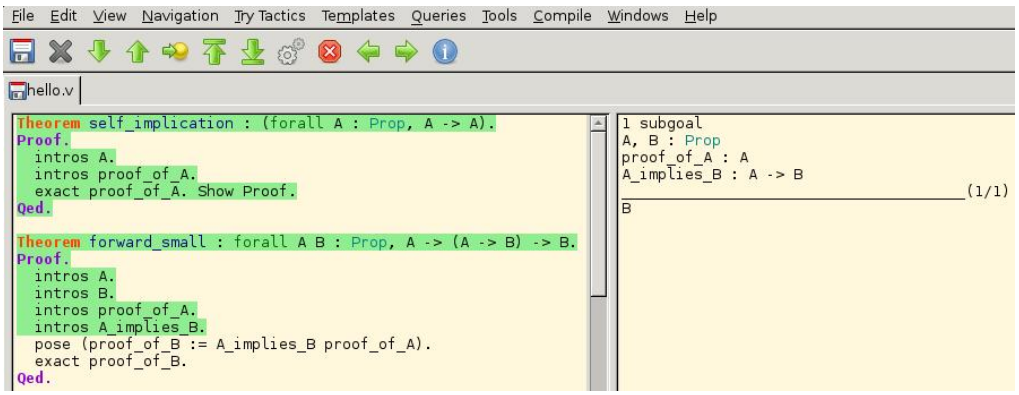



Figure 7: CoqIDE with goal B, hypotheses A and A→B. Green means verified.

We saw a lot of pitfalls earlier that have to be taken care of in some way, and it can be very difficult to reason about everything at once without the formalization model becoming too complicated very quickly. One of the best strategies we have is to split the problem up into multiple pieces, with different models suited to each task. For example, it can often be helpful to prove that no overflow can occur *first*, and then take advantage of this fact by using a less strict model with no maximum $e$ for the rest of the proof.

The things that most formalization models have in common are ways of representing the IEEE-754 specification. For example, the mantissa $m$ can be either an integer or fixed-point number of bounded precision $p$, and the exponent $e$ can be represented as an integer with some offset in its interpretation. $m$ might be representable in exactly $p$ bits of binary precision, or at most $p$, and $e$ may or may not have maximum and minimum values depending on the specific model used. Some models may include special IEEE values `inf` and `NaN`, and some may not. Good models are robust as far as rounding is concerned, and may support multiple options for the different rounding modes defined in the standard. Most have a way of converting back and forth between real and floating point numbers. In fact, most automated proof systems use verified real number libraries for all calculations, which is possible because there is an injective relation between floating-point and real numbers ($\mathbb{F} \subset \mathbb{R}$). As long as rounding operations are performed at the appropriate times, arithmetic proofs for both kinds of arithmetic can be done with the same tools.

Two common metrics are used to measure error introduced by floating-point calculations: absolute error, and relative error. Absolute error is fairly simple, just the difference between the real number result and the floating-point result. Because of the way exact rounding is defined in the standard, we usually represent exact error in terms of *ulp*s, or "units of the last place", or "units of least precision". This measurement depends on the current exponent and precision of the floating-point number. One *ulp* is defined by the value of the least significant digit: when $f = m * \beta^e$ and $m$ uses $p$ bits, $1\ ulp = 1 \times \beta^{e-(p-1)}$. If a number is rounded to the nearest floating-point number from the correct result with exact rounding, the error can be as much as $\frac{1}{2} ulp$.

Relative error is the difference between the real and floating-point number, divided by the real number. One thing to keep in mind is that the relative error corresponding to 1 *ulp* can vary by a factor of $\beta$, depending on the mantissa [5].

$$\frac{1}{2}\beta^{-p} \leq \frac{1}{2} ulp \leq \frac{\beta}{2}\beta^{-p}$$

Usually, absolute error is most useful when we want results to fit in a certain interval.

An absolute error of $\delta$ on an approximation $x$ of some real result $\hat{x}$, for example, means that $x \in [\hat{x} - \delta, \hat{x} + \delta]$, and we can use this strict bound in safety properties. Relative error is more useful in some cases as a measurement irrespective of magnitude, but there is the additional caveat that the relative error to zero is undefined.

One of the most powerful reasoning methods we have for concrete bounds is interval arithmetic, i.e. keeping track of the range of possible value each variable can have throughout the execution of a program. At each assignment operation, we can consider the minimum and maximum values that the operands can have, and use this to come up with a bound for the result accordingly. For instance, if $A \in [-2, 5]$ and $B \in [0, 3]$, then the $A + B \in [-2, 8]$. Calculating these bounds is extremely fast and robust, and we can easily see the effects of the magnitude of the input, for example. Even assuming the maximum rounding error at every stop often gives decent results. However, the results obtained can sometimes be misleading because of the naïve assumption that variables are always independent of each other. One of the challenges we face in the verification is ways to overcome this assumption and prove that dependence can lead to tighter bounds on error.

## 4.3 Flocq

Flocq is the most complete and versatile floating-point formalization library currently available. It has support for both the IEEE binary formats and for reasoning about the underlying bit representation. Types are parameterized by a radix $\beta > 1$, and there are five different models of floating-point numbers with different properties. For example, FLT represents floating-point numbers with limited precision, bounded $e_{\min}$, and gradual underflow, FLX represents those with limited precision and no $e_{\min}$ or $e_{\max}$, and FTZ represents those with bounded precision and $e_{\min}$, but no denormal numbers except zero.

The library also has support for a generic floating-point format that makes it possible to convert between the different models. The library includes model-dependent theorems that have been verified for applicable models. Rounding and *ulp* reasoning are available, and it has been designed for use in Coq.

## 4.4 Gappa

Gappa does not use quite the same integrated interactive approach as Coq, but can be just as useful when used in a similar way. It is designed to run extremely quickly and report the best bounds it can verify. The approach it uses is naïve interval arithmetic, and it simply tries to calculate the closure of a small set of properties iteratively [7]. When it can't improve the bounds beyond a certain threshold anymore, it terminates and reports its answer.

Let's parse what we can see in Figure 8. The first line defines a macro `rnd` for IEEE 32-bit rounding to nearest even. By default, all numbers are treated as reals, so we can use this for strict control of where rounding occurs in our reasoning. In the second block, we define our variables, `x`, `y`, and `z`. The variable `x` is the result of rounding an unbound variable `xx`, i.e. an arbitrary 32-bit floating-point number. The variable `y` is `x(1-x)` when rounding is applied after every operation, and `z` is `x(1-x)` when no rounding is applied.

The third block, enclosed in { }, is the goal we are trying to prove. If the real number `xx` is is in the range $[0, 1]$, we want Gappa to verify the bounds on the variable `y` and the error `y-z`. Instead of providing ranges like `[0,.25]`, we could write `?` instead and Gappa will try to find the best bound it can automatically. However, the bounds

```
@rnd = float<ieee_32,ne>;

x = rnd(xx);
y rnd= x * (1 - x);
z    = x * (1 - x);

{ xx in [0,1] -> y in [0,.25] /\ y-z in [-3b-27,3b-27] }

z -> 0.25 - (x - 0.5) * (x + 0.5);
```

Figure 8: A Gappa script to find bounds for $x(1-x)$ for $x \in [0,1]$

are not always as tight as possible. For example, if the error `y-z` is not specified, Gappa will report bounds of $\pm$`2.98023e-08` instead of $\pm$`2.23517e-08`.

Finally, the last line is a hint provided by the user. Gappa sees the terms `x` and `(1-x)` as independent, and interval arithmetic gives that $([0,1] \times [0,1]) \in [0,1]$. It is, however, able to verify the equivalence to $\left(1/4 - (x-.5)(x+.5)\right)$ mathematically if it is given. By rewriting in this way, Gappa realizes that subtracting a positive quantity from $1/4$ gives an upper bound of 0.25, not 1.

# 5 Frama-C and WP

In order to automate the verification of our `ipol` function, we will annotate the code with proof goals and use the tool `Frama-C` to automate the process as much as possible.

## 5.1 ANSI/ISO ACSL C Specification Language

ACSL, or ANSI/ISO C Specification Language, is a standard for the annotation syntax and rules that we can use to automate the proofs. Most importantly, it can specify functional contracts and logical constructions. The annotations are inserted directly into the code with a special comment syntax, and can then be modified and processed by other programs.

```
/*@ requires \valid(a) && \valid(b);
    ensures A: *a == \old(*b);
    ensures B: *b == \old(*a);
    assigns *a, *b;
*/
void swapf(float *a, float *b) {
    float tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Figure 9: a functional contract that guarantees behaviour and memory safety

In Figure 9, the function contract for `swapf` requires that `a` and `b` are valid memory addresses (have been declared on the heap or stack and not yet been freed or gone out of scope, respectively). It ensures that the values before and after the function call have swapped, and it keeps track of the memory addresses that are written to.

The annotation language treats floating-point numbers as real numbers, only rounding when explicitly stated. Rounding modes can be explicitly chosen, for example `\NearestEven` or `\ToZero`, and `\round_float(rounding_mode m, real x)` will return the rounded result as a real number. Alternatively, C-style casts are equivalent to nearest-even rounding. The specification has support for special IEEE values through built-ins like `\is_NaN`, and even defines `\exact` and `\round_error`. This is all clearly helpful for verification, but unfortunately only a subset of the complete specification has been implemented at this time in WP. Until full support is available, we will have to define our own predicates and be more explicit in our low-level reasoning. If they become available in the future, it is possible that all of our work on bounding values will be redundant, but equally likely that the bounds it is able to prove will be slightly worse because of difficulty in the dependence between variables.

## 5.2 Frama-C/WP Interface

Frama-C can be called as a command-line tool or as an interactive GUI. It can be used to compile and run a program that will crash when the specification/annotations are violated. WP is a plugin for Frama-C that uses weakest precondition calculus to try to prove the annotations correct for all possible runs. It translates the annotations in the C program into an intermediate logical specification language as a preprocessing step, then uses the provers available on the system to attempt to prove the annotations. If the GUI version is invoked, the user has the opportunity to tweak settings and try different provers for each goal.

## 5.3 Frama-C/Jessie

Jessie is an older plugin for Frama-C that works similarly to WP. At the time of writing, it is still able to do some verifications that WP cannot, but the intermediate language that it compiles to is for a tool that is no longer under development. We will use Jessie in our verification, but the goal for the future is to move to WP instead, which is actively being developed and improved. Because it has been around for much longer, most of the literature examples of verified algorithms use Jessie [3, 9].

## 5.4 External Provers and `Why3`

The officially supported external provers are Alt-Ergo and Coq, but there is also an officially-supported prover interface called Why3 that can be used to interface with a large number of other SMT solvers and theorem provers. Those that are compatible with Why3 and include some amount of floating-point reasoning support are:

- Alt-Ergo: SMT solver supported natively by Frama-C
- Coq: native Frama-C support with special tactics, float support through libraries
- Z3: open-source theorem prover from Microsoft Research
- Gappa: special-purpose prover designed for fast interval arithmetic calculations
- CVC4: (another) SMT solver with float support

At the time of writing this, there is an incompatibility in the interface between Why3 compiled with Coq tactic support and the plugin WP, so we will use the individual tools themselves in some places for the verification. The advantage of being able to replay proofs is that when the bug is fixed, it should be possible to load the scripts together for different goals and do automatic verification of everything at once.

# 6 Verifying Interpolation

We will go over the attempts to prove different properties separately and explain steps that were taken for each part. The fully annotated method that is our ideal proof goal is available in the Appendix.

## 6.1 Endpoint Interpolation

For endpoint interpolation, we modify the `ipol` function contract to require special results, `a` and `b` for `t = 0` and `t = 1` respectively. This is achieved simply by conditionally checking for equality, because all numbers involved (0, 1, `a`, `b`) are representable as floating-point numbers. Frama-C implicitly converts the already-rounded values to real numbers for the comparison.

```
/*@ predicate is_finite_flt(real x) =
        -0x1.fffffep+127 <= x <= 0x1.fffffep+127; */
/*@ requires 0 <= t <= 1;
  @ requires a <= b;
  @ requires is_finite_flt(a) && is_finite_flt(b);
  @ requires b - a <= 32767;
  @ ensures endpoint_ipol:
        (t == 1.f ==> \result == b) &&
        (t == 0.f ==> \result == a);
  @ assigns \nothing;
 */
float ipol(float a, float b, float t) {
    float t2 = 1.0f - t;
    //@ assert t == 0.f ==> t2 == 1.f;
    //@ assert t == 1.f ==> t2 == 0.f;
    float left = a*t2;
    float right = b*t;
    float sum = left+right;
    return sum;
}
```

WP is only able to verify "`@ assigns \nothing;`" completely automatically. The `assert` annotations that have been inserted into the body of the function allow it to prove the property `endpoint_ipol`, but it is not able to verify them. So, we turn to Coq in order to attempt to verify the endpoints with human assistance.

In order to verify the annotation "`@ assert t == 0.f ==> t2 == 1.f;`", the following proof goal is generated for Coq:

```
Goal
  forall (r_1 r : R),
  ((r_1 <= r)%R) ->
  ((r <= (( 32767 / 1 )%R + r_1))%R) ->
  ((is_float32 r%R)) ->
  ((is_float32 r_1%R)) ->
  ((P_is_finite_flt r%R)) ->
  ((P_is_finite_flt r_1%R)) ->
  ((((add_float32 ( 1 / 1 )%R%R ( 0 / 1 )%R%R)) = ( 1 / 1 )%R)%R).
```

The syntax is a bit complicated, but we can simplify it by introducing variables and allowing the interface to automatically infer their types. Using the tactic "`intros r1`

r2 H1 H2 H3 H4 H5 H6" gives us something much easier to understand:

```
1 subgoal
r1, r2 : real
H1 : (r1 <= r2)%R
H2 : (r2 <= 32767 / 1 + r1)%R
H3 : is_float32 r2
H4 : is_float32 r1
H5 : P_is_finite_flt r2
H6 : P_is_finite_flt r1
______________________________________(1/1)
add_float32 (1 / 1) (0 / 1) = (1 / 1)%R
```

We are given that r1 and r2 are real number variables, H1-H2 are their bounds, and H3-H6 are assumptions that they can be represented as finite 32-bit floating-point numbers. The problem is automatically reduced to proving that $^1/_1 + {}^0/_1 = {}^1/_1$ under 32-bit floating-point rounding. We need a few basic facts to simplify the expression, so we create three intermediate theorems:

```
theorem inv_identity: Rinv 1 = 1%R.
Proof. firstorder. Qed.
theorem mult_identity: forall r:R, Rmult r 1%R = r.
Proof. firstorder. Qed.
theorem add_identity: forall r:R, Rplus r 0 = r.
Proof. firstorder. Qed.
```

These are formal proofs that $x/1 = x \times 1$, $x \times 1 = x$, and $x + 0 = x$ for all $x$. All three identity theorems are simple enough to be verified automatically using existing axioms and first-order logic, so we do not have to prove them ourselves. Now, we apply them to our goal in order to simplify and hopefully finish the proof. By rewriting the goals with our identities, we are almost able to achieve the proof goal (the complete script is available in the Appendix):

```
  intros r1 r2 H1 H2 H3 H4 H5 H6. unfold add_float32.
GOAL: to_float32 (1 / 1 + 0 / 1) = (1 / 1)%R
  pose (inv_identity) as INV. unfold Rdiv. rewrite INV.
GOAL: to_float32 (1 * 1 + 0 * 1) = (1 * 1)%R
  pose (mult_identity 1) as MULT1. rewrite MULT1.
GOAL: to_float32 (1 + 0 * 1) = 1%R
  pose (mult_identity 0) as MULT0. rewrite MULT0.
GOAL: to_float32 (1 + 0) = 1%R
  pose (add_identity 1) as ADD1. rewrite ADD1.
GOAL: to_float32 1 = 1%R
```

We have shown that it is possible to reduce the problem to proving that the real number 1 is representable as a float! Everything else can be automatically verified by Coq. This is obviously true... unfortunately, this is where the bad news comes in. The libraries used automatically by Coq do not have a definition of what rounding is, just some very basic properties about it. At this time, it is not possible to prove that a specific real number is representable. It should be possible to do something similar to the above proof automatically with Flocq, but until the problem mentioned in 5.4 is fixed, we can instead try reasoning about ipol in Gappa instead:

```
@rnd = float<ieee_32,ne>;
a = rnd(aa);
b = rnd(bb);
t = rnd(tt);
t2    rnd= 1-t;
left  rnd= a*t2;
right rnd= b*t;
sum   rnd= left+right;
real     = (1-t)*a+t*b;
```

This is a translation of `ipol` into the language of Gappa. Some of the tools available, like Why3, are able to automatically transform the function from C to Gappa syntax automatically. Now, we can choose to either reason about the absolute error, or the relative error:

```
# absolute error
error    = sum - real;
{
aa in [-0x1.fffffep+127,0x1.fffffep+127] /\
bb in [-0x1.fffffep+127,0x1.fffffep+127] /\
tt in [0,0] ->
    sum in ? /\ error in ?
}
```

```
# relative error
rel_err  = error / real;
{
aa in [0x1p-149,0x1.fffffep+127] /\
bb in [-0x1.fffffep+127,0x1.fffffep+127] /\
tt in [0,0] ->
    sum in ? /\ error in ? /\ rel_err in ?
}
```

The full Gappa script is also available in the Appendix. We have to include the minimum and maximum possible float values for `aa` and `bb`, and Gappa reports an error bound of $\pm 2^{103}$, or about $\pm 3.4 \times 10^{38}$. This is helpful for inputs of large magnitude, but not for inputs of small magnitude. If we split `aa` into zero and non-zero cases, we get either no error or a relative error of $\pm 1 \times 2^{-24}$. In 24-bit precision, that is $^1/_2$ulp, and nearly the best bound we can hope for. Gappa is able to prove that `sum` has exactly the same bound as `aa`, but it is unfortunately not smart enough to stop adding rounding errors when a representable number is rounded again.

We have not been able to automatically verify Endpoint Interpolation using WP or Gappa, but this is one case where Jessie is still superior to the newer tools still under development. If we pass the original ACSL goal-annotated function for `ipol`, Jessie is able to verify using Why3 and Alt-ergo that the endpoint interpolation is exact for the values `t = 0` and `t = 1`. We have shown that proving it is feasible to check automatically and quickly with three different methods, one of which requires no human input. For full automation in WP, it is likely that all we need is the bug-fix that allows Why3 to use its floating-point library or interface with Coq-Flocq.

## 6.2 Bounded Interpolation

In order to prove that `ipol` is bounded by `a` and `b`, we insert a new `@ensures` clause to check the bounds of `\result`. However, since we have already proven exact results for endpoints, it might make sense to split into two cases here, $t = 0$ and $t > 0$. So, we include both to see what we are able to verify.

```
/*@ predicate is_finite_flt(real x) =
        -0x1.fffffep+127 <= x <= 0x1.fffffep+127; */
/*@ requires 0 <= t <= 1;
  @ requires a <= b;
  @ requires is_finite_flt(a) && is_finite_flt(b);
  @ requires b - a <= 32767;
  @ ensures t_eq_zero: t == 0.f ==> \result == a;
  @ ensures t_gt_zero: t > 0.f ==> \result >= a;
  @ ensures bounded_ipol: a <= \result <= b;
  @ assigns \nothing; */
float ipol(float a, float b, float t) {
    float t2 = 1.0f - t;
    //@ assert (float)(t+t2) == 1.0f;
    float left = a*t2;
    float right = b*t;
    float sum = left+right;
    return sum;
}
```

Frama-C/WP is again unable to prove the bounds by itself. If we try to do a human-assisted proof with Coq, we run into the same problem we did for endpoint interpolation, i.e. that rounding is not implemented in the default library. We know that Gappa is designed for proving bounds, so we can try using that instead. We will start with smaller bounds to see what Gappa is able to prove, then try again with the actual input bounds of representable 32-bit floating-point numbers if we are successful. The goal is the following:

```
{ aa in [-100,100] /\ bb in [-100,100] /\ t in [0,1] ->
    sum in ? /\ real in ? /\ error in ? }
```

Gappa reports that `sum` and `real` are both in the range [-200,200]. This is a result of the independence assumption we mentioned earlier. Gappa knows that both `t` and `1-t` have bounds of `[0,1]`, so it can only tell us that by interval arithmetic,

$$[0,1] * [-100,100] + [0,1] * [-100,100] \in [-200,200]$$

It has no concept of the fact that when `a*(1-t)` is very large in magnitude, `b*t` must be very small. We can force it to make tighter bounds by splitting up `t` into multiple equal-sized intervals, by passing it user hints.

```
$ t in 2;    # splits the error in 2:       [-150,150]
$ t in 4;    # splits the error in 2 again: [-125,125]
$ t in 2048; # splits the error even more:  [-101.025,101.025]
```

Splitting `t` into 2048 equal-sized slices takes only a few seconds to run, but each time we split the interval size if half, it takes twice as long. To get the bound we want, we

would have to keep splitting intervals for `t` an unfeasible number of times. Clearly, this won't work with much larger numbers.

Luckily, there is a second way we can tell Gappa to rewrite the interpolation formula. In the realm of $\mathbb{R}$, the following is true:

$$(1-t)a + tb = a + t(b-a)$$

Note that this is *not* always true of floating-point numbers because of rounding errors and overflow. However, we can provide a "user hint" to tell Gappa that `real` is equivalent to this $a$-form and the similar $b$-form.

```
real -> a - t*(a-b);
real -> b - (1-t)*(b-a);
```

At this point, Gappa still reports incorrect bounds. However, we see mathematically that, if $a > b$, then $(a-b) > 0$, and thus $a - t*(a-b)$ is at most `a`, and vice versa. We can force Gappa to consider this case split in the relative value of `a` and `b` by adding a separate premise:

```
{ aa in [-100,100] /\ bb in [-100,100] /\
t in [0,1] /\ (a-b >= 0 \/ b-a >= 0)            # (a >= b) || (b >= a)
->
sum in ? /\ real in ? /\ error in ?  }
```

Finally, we get the correct bounds `[-100,100]` for `real`. If we use the input restrictions of what is representable, i.e. the range given by `±0x1.fffffep+127`, we get the correct bound for `real` but an unrepresentable bound for `sum`: $\pm 2^{128}$. If exponents were unbounded, this would be the next-highest floating-point number after `FLT_MAX`. In our specific scenario, because the inputs have no size bound other than representability, proving the absence of overflow will be enough to prove our bound. However, in the general case a more careful proof would be required. The full Gappa script for this section is available in the Appendix.

We are not done! With Gappa, we are able to prove results based on the intervals of possible inputs, but there is no way to relate the result back to a specific input. If all we know is that $\texttt{a} \in [-100, 100]$, it could be the case that $\texttt{a} = 0.375$, and $\backslash\texttt{result} \geq -100$ is not useful. The problem is this: disregarding `inf` and `NaN`, floating-point addition and multiplication are monotonic functions. That is,

$$\begin{aligned} &\forall_{a,b,c} \quad a \geq b \rightarrow a \oplus c \geq b \oplus c \\ &\forall_{a,b,c} \quad a \geq b \rightarrow (c \geq 0 \rightarrow a \otimes c \geq b \otimes c) \end{aligned}$$

With nearest/ties-to-even rounding, it is the case that $(1 \ominus t) \oplus t = 1$ because the rounding steps will always counteract each other. But when we scale the two subterms by different values, we no longer have this guarantee:

$$((1 \ominus t) \otimes a) \oplus (t \otimes b) \overset{?}{\in} [a, b]$$

In `ipol`, we have a term $(1 \ominus t)$ that is monotonically *decreasing* as $t$ increases. It might be possible that the error introduced by rounding the compound terms $((1 \ominus t) \otimes a)$ and $(t \otimes b)$ can take us out of the range $[a, b]$. As it turns out, this is the case! After some failed attempts to do a proof with Jessie and Coq, it seemed like a good idea to look for a counterexample.

$$\text{initial state}: \begin{cases} a = \texttt{-0x1.0297dep+13} \\ b = \texttt{-0x1.0297dcp+13} \\ t = \texttt{ 0x1.93e896p-19} \end{cases}$$

$$\text{result}: \begin{cases} 1 \ominus t = \texttt{0x1.ffff9cp-1} \\ b \ominus a = \texttt{0x1.000000p-10} \\ \texttt{ipol(a,b,t)} = \texttt{-0x1.0297ep+13} \end{cases}$$

In this counterexample, we see that all our preconditions are met:

- $t \in [0, 1]$: the real-number interpolation goal is bounded by $a$ and $b$
- $b - a \in [0, 32767]$: $a$ and $b$ differ by at most `int16_t_MAX`, and $a \leq b$
- $a$ and $b$ are finite, representable, 32-bit floating-point numbers

It is worth pointing out that there are not even any denormal numbers here. Not only did monotonicity of `ipol` fail, but it failed near the endpoints and went outside of the expected bounds. That is a fail for our desired **Bounded Interpolation** property. Luckily, this does not make our specification completely useless. We know that the real result of linear interpolation *does* reside in the bounds $[a, b]$, so a good result on **Bounded Error** (let's call it $\delta$) would still give us reasonable bounds $[a - \delta, b + \delta]$. Note also that in this counterexample, the error is still within a *ulp*. In the context of Scania's library function, many of the degenerate cases ($a = b, t \in (0, 1)$) where this can happen are avoided, but we still want **Bounded Error** for a good final result.

## 6.3 Order Invariance/Sign Invariance

Now, we want to see if this mathematical property of interpolations holds. Once again, we start with an ACSL specification of the properties we want to prove. In this case, because we will be comparing the results of multiple different calls to `ipol` with different parameters, it makes sense to start by creating a *logical* function and proving its equivalence. It is impossible to relate separate function calls within the function specification itself because of its limited scope.

```
/*@ logic real float_ipol(float a, float b, float t) =
        0+(float)(
            (float)(
                (float)(1-t)*a
            ) + (float)(t*b)
        ); */
/*@ ensures \result == float_ipol(a,b,t); */
float ipol(float a, float b, float t) {
    float t2 = 1.0f - t;
    float left = a*t2;
    float right = b*t;
    float sum = left+right;
    return sum;
}
```

The reason for the "$0 + \ldots$" is to make Frama-C treat the result as a real value for the comparison, as it does implicitly with the return value of a function. Frama-C/WP is able to prove the equivalence of the two functions automatically, so all we have to do is write out our ACSL specification using this new format.

```
/*
@ logic real real_ipol(real a, real b, real t) =
      (1-t)*a + t*b;
@ lemma sign_order_invariance_real:
    \forall float a, float b, float t;
    real_ipol(a,b,t) == - real_ipol(-b,-a,(1-t));

@ logic real float_ipol(float a, float b, float t) =
    0+(float)(
        (float)(
            (float)(1-t)*a
        ) + (float)(t*b)
    );
@ lemma sign_order_invariance_float:
    \forall float a, float b, float t;
    (
        a <= b &&
        0 <= t <= 1 &&
        is_finite_flt(a) &&
        is_finite_flt(b) &&
        b-a <= 32767
    ) ==> (
        \let neg_b = (float)(-b);
        \let neg_a = (float)(-a);
        \let inv_t = (float)(1-t);
        float_ipol(a,b,t) == -float_ipol(neg_b,neg_a,inv_t)
    );
*/
```

In our purely logical specification, we have two goals. The first is to prove that in `real_ipol`, with real numbers and no rounding, both sign and order invariance hold. This can be verified automatically using Frama-C/WP. The second is to prove that the same holds true for `float_ipol`. Gappa cannot help in this case because we want to prove that properties hold *for each value*, so ranges will not work. Our best option is to use Frama-C/Jessie/Coq once more.

Jessie has a more complete set of axioms available, but not enough to do the whole proof automatically. Most of it can be done with rewriting rules by moving around the negative signs and cancelling them out, but there are no axioms that tell us rounding is not affected by the sign of the argument. Under "round to nearest, ties to even", this is the case because the sign bit cannot affect which adjacent number is nearest or the parity of the mantissa. So, we will assume this for now and try to prove it later if the proof goes through.

```
Axiom Ropp_round: forall r:R,
  value (round_logic NearestTiesToEven (Ropp r)) =
  Ropp (value (round_logic NearestTiesToEven r)).
```

In Jessie's library, `single` is an alias for `float32`, and `value` is a function converting `single` $\rightarrow$ `real`. What we are stating is that the real result of rounding $-r$ (ties to nearest even) is equal to the additive inverse of rounding $r$ by itself. We can apply this axiom in a script:

```
intros a_2 b_2 t_2 (h1,(h2,(h3,(h4,(h5,h6))))).
unfold float_ipol. rewrite Rplus_0_l. rewrite Rplus_0_l. rewrite Ropp_round.
rewrite Ropp_round. rewrite <- Ropp_mult_distr_r. rewrite <- Ropp_mult_distr_r.
rewrite Ropp_round. rewrite Ropp_round. rewrite <- Ropp_plus_distr.
rewrite Ropp_round. rewrite Ropp_involutive.
rewrite (Round_logic_def NearestTiesToEven (value b_2) (Bounded_real_no_overflow
    NearestTiesToEven (value b_2) (Bounded_value b_2))).
rewrite (Round_logic_def NearestTiesToEven (value a_2) (Bounded_real_no_overflow
    NearestTiesToEven (value a_2) (Bounded_value a_2))).
rewrite (Round_value NearestTiesToEven a_2).
rewrite (trans _ (value (round_logic NearestTiesToEven (value t_2 * value b_2)))).
```

With these rewrite rules and axiom instantiations we get an almost-complete goal:

```
value (round_logic NearestTiesToEven (value
    (round_logic NearestTiesToEven (value t_2 * value b_2)) +
    value (round_logic NearestTiesToEven (value
        (round_logic NearestTiesToEven (1 - value t_2)) * value a_2))))
=
value (round_logic NearestTiesToEven (value
    (round_logic NearestTiesToEven (value
        (round_logic NearestTiesToEven (1 - value
            (round_logic NearestTiesToEven (1 - value t_2)))) * value b_2)) +
    value (round_logic NearestTiesToEven (value
        (round_logic NearestTiesToEven (1 - value t_2)) * value a_2))))
```

The problem is reduced to showing that $1 \ominus (1 \ominus t_2)) = t_2$. Unfortunately, as with the last property, this is not always the case. Once again, by using semi-automated formal proof techniques, we are able to find where a property fails to hold, and use that to search for a counterexample. In this case it is **Order Invariance** that does not hold and **Sign Invariance** that does, but we are unable to prove one without the other because of the precondition that $b \geq a$. This failure does not occur when $t_2 \geq \frac{1}{2}$, using the Sterbenz lemma [5]. However, it only dictates exact rounding for any two numbers within a power of 2 of each other, so we have no such guarantee when $t_2 < \frac{1}{2}$.

## 6.4 Absence of Overflows

In order to prove the absence of overflows, we need to look at the value of every intermediate calculation both before and after rounding. Because `FLT_MAX` has an odd mantissa, and rounding ties go to even, we need a guarantee that the result of an intermediate calculation before rounding never becomes `0x1.fffffp+127` or higher (and the same applies at the negative end). This is exactly $^1/_2 ulp$ higher than `FLT_MAX`. Gappa is the most useful tool for bounds like this, but, as we know, it is unable to find the tightest bounds when terms are interdependent, and it adds rounding error even for exact values. We do not expect to be able to prove this goal with Gappa, but it is interesting to see what results we can get anyways.

In Figure 10, we reduce the bounds of `a` and `b` by one *ulp* because we want to get a result of "is representable", and we know that Gappa is not exact enough to produce this if the input is all representable numbers because of rounding error. We have to use the same hints as before to overcome the dependency effect. Even with the user

```
{
aa in [-0x1.fffffcp+127,0x1.fffffcp+127] /\
bb in [-0x1.fffffcp+127,0x1.fffffcp+127] /\
tt in [0,1] /\ (b-a >= 32767)
->
    left + right in [-0x1.fffffep+127,0x1.fffffep+127]
    /\ result in [-0x1.fffffep+127,0x1.fffffep+127]
}
real -> a - t*(a-b);
real -> b - (1-t)*(b-a);
```

Figure 10: Approximating overflow with Gappa. Full script in Appendix

annotations, Gappa is not able to bound the results to representable numbers. This is a result of the way Gappa calculates these bounds in the first place.

As mentioned before, Gappa starts with a small set of properties for each variable or symbol, and iteratively tightens the bounds where it can. In order to ensure termination, it discards results if the new result is not significantly better than the old one. The default threshold is 0.01, meaning if a property is not improved by at least 1%, it is not updated. We can override this with the command-line parameter `-Echange-threshold=0`. Finally, using this option, Gappa is able to prove that for `a` and `b` strictly less than `FLT_MAX`, the result is representable. Unfortunately, it is not capable of proving the absence of overflow when `a` and `b` can each be *equal* to `FLT_MAX`.

Now, it is time to take a step back and look at what our restrictions are one more time. Let us assume for now that $a \neq b$. We have not considered that if adjacent values are at most 32767 (`int16_t_MAX`) apart, the intervals for `a` and `b` are limited by the precision! In fact, `0x1.fffffep+38` is the highest floating-point number for which adding 32767 and rounding changes the result. So, instead of using the bounds of what is representable, we can use $\pm$`0x1p+39` instead! This is enough for Jessie+Gappa to automatically prove that no overflow can occur. So, it is difficult to prove bounds in the most extreme cases, but with stricter preconditions it is no problem.

## 6.5 Bounded Error

Due to time constraints and software bugs, it was not possible to complete the automatic proof of bounded error of `ipol`. Gappa is able to provide good interval bounds, but unable to provide bounds related to specific input values. Jessie has an incomplete specification of what rounding is, and WP is still waiting for a compatibility fix. Unfortunately, it does not seem possible to automatically calculate useful bounds at this time without a human-driven formal proof. The most promising strategy is to write a proof with Flocq to be checked automatically, but time restrictions prevented this for the current step of the verification project. So instead, we will go over what this kind of proof would look like formally, and what axioms available in Flocq. Examples of such proofs are given in the Flocq repository as a proof of concept [9].

The proof strategy is similar to interval arithmetic, but starting with an arbitrary singleton interval for each variable and assuming the worst error at every step to avoid dependency effect problems. We can derive a formula to find the value of 1 *ulp*, and use its monotonicity to create upper bounds at each step where rounding error can be introduced. The error introduced at every step is bounded by $\frac{1}{2}$ *ulp* of the real answer under IEEE-754 exact rounding, and the size of a *ulp* depends in turn on the interval of possible inputs for each operand. Because we have four rounded operations to introduce

error, we can reason about the error of each, $e_1 \dots e_4$:

$$\texttt{ipol(a, b, t)} = \underbrace{\overbrace{(\underbrace{(1 \ominus t)}_{e_1} \otimes a)}^{e_2} \oplus \overbrace{(t \otimes b)}^{e_3}}_{e_4}$$

Flocq has a theorem, `error_le_half_ulp_round`, which states that $(\circ(x) - x) \leq \frac{1}{2} ulp(\circ(x))$. This will give us the tightest bounds for each rounding interval that we can get without complicated bit-level proofs. The Flocq formalization also has the following theorems among many others:

| **Theorem** | **Meaning** | **Bounds** |
|---|---|---|
| `ulp_ge_ulp_0` | $ulp(x) \geq ulp(0)$ | $\mathbb{R}$ |
| `ulp_le_abs` | $ulp(x) = ulp(\lvert x \rvert)$ | $\mathbb{F} \setminus \{0\}$ |
| `ulp_le_id` | $ulp(x) \leq \lvert x \rvert$ | $\mathbb{F} \setminus \{0\}$ |
| `ulp_le` | $\lvert x \rvert \leq \lvert y \rvert \rightarrow ulp(x) \leq ulp(y)$ | $\mathbb{R} \times \mathbb{R}$ |

Of particular importance is `ulp_le`, which guarantees the monotonicity of *ulp* under absolute value. We will have to split up the *ulp* in our proof into normal and denormal numbers because of $e_{min}$, and we will re-derive `ulp_le_id` for denormal numbers and show a stronger theorem for normal numbers. Now, we need a formula to calculate the *ulp* of a real number $x$ in 32-bit float format:

$$ulp(x) = 2^{\texttt{max}(-149, \lfloor \log_2(\lvert x \rvert) \rfloor - 23)}$$

The first step would be to prove this formally in Flocq, but we will not do so now. We take this formula and use it to prove an upper bound that we can use for $ulp(x)$ later.

**Lemma 1**: Bounding *ulp*: $ulp(x) \leq \begin{cases} 2^{-149} & \lvert x \rvert < 2^{-125} \\ 2^{\lfloor \log_2 \lvert x \rvert \rfloor - 23} & \lvert x \rvert \geq 2^{-125} \end{cases}$

$$\begin{aligned} ulp(x) &= 2^{\texttt{max}(-149, \lfloor \log_2(\lvert x \rvert) \rfloor - 23)} \\ &= \begin{cases} 2^{-149} & \lfloor \log_2 (\lvert x \rvert) \rfloor - 23 \leq -149 \\ 2^{\lfloor \log_2(\lvert x \rvert) \rfloor - 23} & \lfloor \log_2 (\lvert x \rvert) \rfloor - 23 > -149 \end{cases} \end{aligned}$$

We choose to use the form $2^{-149}$ when the two arguments to `max` have the same value. Now, we simplify the conditions to something easier to understand:

$$\begin{aligned} \lfloor \log_2 (\lvert x \rvert) \rfloor - 23 &\leq -149 & \\ \lfloor \log_2 (\lvert x \rvert) \rfloor &\leq -126 & \text{add 23} \\ \log_2 (\lvert x \rvert) &< -125 & \text{by definition of } \lfloor \rfloor \\ 2^{\log_2(\lvert x \rvert)} &< 2^{-125} & 2^x \text{ is monotonically increasing} \\ \lvert x \rvert &< 2^{-125} & \text{property of logarithms} \end{aligned}$$

This is more meaningful. When $x < 2^{-125}$, $e = e_{min}$, so of course $ulp(x) = 2^{-149}$.

$$ulp(x) = \begin{cases} 2^{-149} & \lvert x \rvert < 2^{-125} \\ 2^{\lfloor \log_2(\lvert x \rvert) \rfloor - 23} & \lvert x \rvert \geq 2^{-125} \end{cases}$$

Using the same properties for the *ulp* formula in the second case, we have our bound:

$$ulp(x) \leq \begin{cases} 2^{-149} & |x| < 2^{-125} \\ 2^{-23} \cdot |x| & |x| \geq 2^{-125} \end{cases}$$

The next step almost the same, but for going back from real numbers to *ulp*'s.

**Lemma 2**: bounding reals: $2^{-24}|x| \leq ulp(x)$

| | |
|---|---|
| $2^{-24}\|x\| \leq 2^{-149}$ | **if** $\|x\| < 2^{-125}$ |
| $\log_2 \|x\| \leq \lfloor \log_2 \|x\| \rfloor + 1$ | property of $\lceil\ \rceil$ and $\lfloor\ \rfloor$ |
| $\frac{1}{2}\|x\| \leq 2^{\lfloor \log_2 \|x\| \rfloor}$ | |
| $2^{-24}\|x\| \leq 2^{\lfloor \log_2 \|x\| \rfloor - 23}$ | **if** $\|x\| \geq 2^{-125}$ |
| $2^{-24}\|x\| \leq ulp(x)$ | (both cases handled) |

Now we have useful bounds for 1 *ulp* that do not depend on the bit representation. They are slightly less precise than before, but easier to generalize. They are also somewhat intuitive. In **Lemma 1**, multiplying by $2^{-23}$ is the same as shifting the implicit leading 1 of a normalized floating-point number through all 23 precision bits of $m$ to the last place. For a power of two, this is 1 *ulp* exactly. For numbers in between, it is slightly larger because of the non-zero mantisa bits (i.e. it is an upper bound). In **Lemma 2**, the leading bit is shifted even farther, out of the representable range, and thus is at most 1 *ulp*.

**Theorem 1**: Bounding `ipol` error

We are guaranteed that the error of a single operation is at most one *ulp* from the mathematically exact result, i.e. rounded correctly.

$$e_1 \leq \frac{1}{2} ulp(1-t)$$

We apply Lemma 1, and move the $\frac{1}{2}$ into the exponent:

$$\leq \begin{cases} 2^{-150} & (1-t) < 2^{-125} \\ 2^{-24} \cdot (1-t) & (1-t) \geq 2^{-125} \end{cases}$$

We know $t \in [0,1]$ and $t \in \mathbb{F}$, so we can use the the properties of 32-bit precision and rounding to simplify the conditions. $1 - t < 2^{-125} \equiv t > 1 - 2^{-125}$, and the only representable $t$ in this range is 1:

$$= \begin{cases} 2^{-150} & t = 1 \\ 2^{-24} \cdot (1-t) & t \neq 1 \end{cases}$$

We already showed with endpoint interpolation that the error is 0 when $t = 1$, so we assume the new bounds $t \in (0,1)$ and simplify:

$$e_1 \leq 2^{-24} \cdot (1-t)$$

We use a similar process for $e_2$, but using the maximum error bound for $e_1$ to increase the absolute value of the result to a new upper bound:

$$
\begin{aligned}
e_2 &\leq \frac{1}{2} ulp\big((1 - t + e_1) \cdot |a|\big) \\
e_2 &\leq \frac{1}{2} ulp\big((1 - t + 2^{-24} \cdot (1 - t)) \cdot |a|\big) \\
e_2 &\leq \frac{1}{2} ulp\big((1 + 2^{-24}) \cdot (1 - t) \cdot |a|\big) \\
&\leq \begin{cases} 2^{-150} & (1 + 2^{-24}) \cdot (1 - t) \cdot |a| < 2^{-125} \\ 2^{-24} \cdot (1 + 2^{-24}) \cdot (1 - t) \cdot |a| & (1 + 2^{-24}) \cdot (1 - t) \cdot |a| \geq 2^{-125} \end{cases}
\end{aligned}
$$

Then, we do the same with $e_3$:

$$
\begin{aligned}
e_3 &\leq \frac{1}{2} ulp(t \cdot |b|) \\
&\leq \begin{cases} 2^{-150} & t \cdot |b| < 2^{-125} \\ 2^{-24} \cdot (t \cdot |b|) & t \cdot |b| \geq 2^{-125} \end{cases}
\end{aligned}
$$

Finally, we apply the error formula for $e_4$, again adding the maximum error bounds from previous calculations:

$$
e_4 \leq \frac{1}{2} ulp\big(|a \cdot (1 - t) + b \cdot t| + e_2 + e_3\big)
$$

Now, we have two possibilities each for $e_2$ and $e_3$, so we will split the proof of error for $e_4$ into four cases.

**Case 1:** $(1 + 2^{-24}) \cdot (1 - t) \cdot |a| \geq 2^{-125}$ and $t \cdot |b| \geq 2^{-125}$

$$
\begin{aligned}
e_2 &\leq 2^{-24} \cdot (1 + 2^{-24}) \cdot (1 - t) \cdot |a| \\
e_3 &\leq 2^{-24} \cdot (t \cdot |b|)
\end{aligned}
$$

We know that $t \in (0, 1)$, and so $(1 - t) \in [0, 1]$ as well. Therefore, $t \cdot |b| < |b|$. Similarly, we know that $(1 + 2^{-24}) \cdot (1 - t) \cdot |a| < (1 + 2^{-24}) \cdot |a|$.

$$
\begin{aligned}
e_2 &< 2^{-24} \cdot (1 + 2^{-24}) \cdot |a| \\
e_3 &< 2^{-24} \cdot |b|
\end{aligned}
$$

We split apply Lemma 1 to $e_4$:

$$
e_4 \leq \begin{cases} 2^{-150} \\ 2^{-24} \cdot \big(|a \cdot (1 - t) + b \cdot t| + e_2 + e_3\big) \end{cases}
$$

$2^{-150}$ is not representable, so the error is always within 1 *ulp* and we disregard it. We expand the second case using the bounds for $e_2$ and $e_3$:

$$
\begin{aligned}
e_4 &< 2^{-24} \cdot \Big(\big|a(1 - t) + b(t)\big| + 2^{-24} \cdot (1 + 2^{-24}) \cdot |a| + 2^{-24} \cdot |b|\Big) \\
&= 2^{-24} \cdot \big|a(1 - t) + b(t)\big| + (2^{-48} + 2^{-72}) \cdot |a| + 2^{-48} \cdot |b|
\end{aligned}
$$

Now, we apply lemma 2 to get an upper bound:

$$e_4 \leq ulp(\texttt{real\_ipol}(a,b,t)) + \left((2^{-48} + 2^{-72}) \cdot |a|\right) + \left(2^{-48} \cdot |b|\right)$$
$$e_4 \leq ulp(\texttt{real\_ipol}(a,b,t)) + \left(2^{-47} \cdot |a|\right) + \left(2^{-48} \cdot |b|\right)$$
$$e_4 \leq ulp(\texttt{real\_ipol}(a,b,t)) + 2^{-23}ulp(a) + 2^{-24}ulp(b)$$

**Case 2:** $(1 + 2^{-24}) \cdot (1 - t) \cdot |a| < 2^{-125}$ and $t \cdot |b| < 2^{-125}$

$$e_2 \leq 2^{-150}$$
$$e_3 \leq 2^{-150}$$

We use the same steps as case 1:

$$e_4 \leq ulp(\texttt{real\_ipol}(a,b,t)) + 2^{-24} \cdot 2^{-150} + 2^{-24} \cdot 2^{-150}$$

The error terms are less than the minimum *ulp* in 32-bit floating-point format, so we can upper bound it using the same values as case 1:

$$e_4 \leq ulp(\texttt{real\_ipol}(a,b,t)) + 2^{-23}ulp(a) + 2^{-24}ulp(b)$$

**Case 3** and **Case 4** are just a combination of cases 1 and 2, and are omitted.

**Result**: $e_4 \leq ulp(\texttt{real\_ipol}(a,b,t)) + 2^{-23}ulp(a) + 2^{-24}ulp(b)$

We aren't able to prove exact rounding, of course; this was expected because of the **Bounded Interpolation** counterexample that was off by slightly more than 1 *ulp*. However, it is a useful measure of the error in terms of the inputs. In most cases, the *ulp*'s of $a$, $b$, and the result will be of similar magnitudes to each other, so the constants on $ulp(a)$ and $ulp(b)$ mean that the error is usually at most 1 or 2 *ulp*s from the real answer. However, there are some extreme cases where $ulp(b)$ is more that 24 orders of magnitude larger than $ulp(\texttt{real\_ipol}(a,b,t))$, and the error of the answer becomes more dependent on $b$ than the real interpolation value. An example of this might be when $a \approx 0$, $t \approx 0$, and $b \approx$ `int16_t_MAX`. This is, however, an expected result of combining floating-point numbers that are extremely different in magnitude, and we have still managed to prove a useful bound for any valid inputs. The **Bounded Error** property is a success for now, and future work includes verifying the proof more formally and automatically in Coq.

# 7 Results of the Case Study

## 7.1 What was Successful

We were able to use the tools available to prove properties of a floating-point function for both specific inputs and ranges of inputs. A summary of the results is available in Table 2. **Endpoint Interpolation** is an example of proving properties about specific inputs to the function, and was handled completely automatically using Jessie. This type of function contract is scalable and has easily automatable proofs.

| | |
|---|---|
| **Endpoint Interpolation** | Proved with Jessie |
| **Bounded Interpolation** | Counterexample found with **Bounded Error** |
| **Order Invariance** | Does not always hold |
| **Sign invariance** | Not testable without **Order Invariance** |
| **Absence of Overflows** | Overflow does not occur |
| **Bounded Error** | Proved a fairly tight bound |
| Automation | Partial automation, not integrated |

Table 2: Results of the verification case study

Some properties, like **Absence of Overflows**, are easy to automate with tighter input bounds to a function, but much more difficult to generalize with less tight restrictions. In a large code base with lots of interdependent functions, there is a trade-off to be balanced between expeditious integration/testing and more useful safety bounds. It may take a huge time investment to prove the tightest possible bounds possible for each function, but the highest level of security and certainty can be achieved by doing so. However, the amount of time spent proving the bounds can be a drain on resources, as it can be a very slow process that largely cannot by automated.

Finally, some properties like **Bounded Interpolation** have proof dependencies (**Bounded Error**), and cannot easily be automated on their own. **Bounded Error** is the most important property of all for floating-point functions that approximate real-number operations, but it is also the most difficult to prove. Even with the high degree of certainty we can get using a formal written proof, there is still room for improvement via automation. We are not yet at the point where a full industrial floating-point verification is feasible, but with continued efforts and tool development it may become a possibility in the future.

### 7.2 What was not...

The only property we attempted to prove that downright failed was **Order Invariance**. It is an example of a property that holds for real numbers, but does not carry over into floating-point calculations. However, it is not immediately obvious that this property would fail without a great deal of prior knowledge and insight. Even though we were not able to prove that the property holds, we were able to use the attempted proof as a way to diagnose where exactly the problem was.

**Bounded Interpolation** is our greatest success in this regard. Although we were not able to prove the bounds we were looking for, the failed proof led straight to a counterexample and a better understanding of why the property does not hold. The result being technically out of the bounds looks bad from a formal specification standpoint, but we are still able to reason about the overall error from the real value that we *know* is bounded, and derive useful safety properties. The conclusion we can draw from this is that even if we fail some of our proof goals, we can still use those failures to strengthen what we have and end up with a more correct and useful specification.

## 8 Future Work to be Done

Due to time constraints and bugs in some of the tools, we were not able to combine the proofs of our specification into one single workflow. The most important next step is integration into one tool or interface, which may, but not necessarily, end up being Frama-C/WP. There is also a human component to the interface that is not automatable,

namely verifying that the hardware and compiler adhere to the strict specification used to verify the software. This must be verified for every platform, and every time something changes. If this can all be achieved, there needs to be a way of transforming high-level safety properties into the specifications to be automatically checked. It needs to be accessible so that somebody unfamiliar with the code itself is still able to use the tools in an effective way.

Another big hurdle is the machine-verifiable formal proof of the error bounds in Section 6.5. The most powerful formalization tools available for these kinds of proofs, like Flocq, have a level of detail so high that they are also the most difficult and time-comsuming to use. Over the course of verifying a larger code base, it will be necessary to build up another library of theorems to simplify the process, and possible even another interface on top of it to translate high-level ideas into the low-level concepts required to make the proofs work.

# References


[1] Gurov, Lidström, Nyberg, Westman. Deductive Functional Verification of Safety-Critical Embedded C-Code: An Experience Report. Critical Systems: Formal Methods and Automated Verification. ISBN 9783319671130

[2] Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabi. IMTEC-92-26: published Feb 4, 1992. Publicly Released: Feb 27, 1992.

[3] Sylvie Boldo. Formal Verification of Programs Computing the Floating-Point Average. Michael Butler; Sylvain Conchon; Fatiha Zaidi. 17th International Conference on Formal Engineering Methods, Nov 2015, Paris, France. Springer, 9407, pp.17-32, 2015.

[4] Clément Fumex, Claude Marché, Yannick Moy. Automating the Verification of Floating-Point Pro-grams. Andrei Paskevich and Thomas Wies. 9th Working Conference on Verified Software: Theories,Tools and Experiments, Jul 2017, Heidelberg, Germany. Springer, 10712, 2017, Lecture Notes inComputer Science

[5] Goldberg, David. What Every Computer Scientist Should Know About Floating-point Arithmetic. ACM Comput. Surv. March 1991. Vol 23, No 1, issn 0360-0300, pgs5-48. DOI 10.1145/103162.103163

[6] John Harrison. Floating-point Verification Using Theorem Proving. Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer Communication, and Software Systems, SFM 2006, colume 3965 of Lecture Notes in Computer Science, pg211-242

[7] Dinechin F., Lauter C., Melquiond G. Certifying floating-point Implmentations using Gappa. CoRR abs/0801.0523, 2008.

[8] David Monniaux.The pitfalls of verifying floating-point computations.ACM Transactions on Programming Languages and Systems (TOPLAS), ACM, 2008, 30 (3), pp.12. 10.1145/1353445.1353446.

[9] Boldo S., Melquiond G. Computer Arithmetic and Formal Proofs. ISBN 9780081011706. ISTE Press - Elsevier, 16 November 2017.

[10] Dawson, Bruce. Floating-Point Determinism. Random ASCII. published July 16, 2013.

[11] Mike Nahas. The Coq Proof Assistant. 10 Oct 2010, version 1.0

# 9 Appendix

```
tF32 Util_1dIpol_F32(const tF32 * const axis_aF32,
                           tS32         lenX_S32,
                     const tF32 * const vector_aF32,
                           tF32         valX_F32)
{
    tS32 ct_xRight_S32 = 0; /* Index for search along axis_x */
    tS32 ct_xLeft_S32;      /* Index variable for referencing ct_xRight_S32-1. */
    tF32 xdiff_F32;         /* xdiff_F32: difference between two axis_x values */
    tF32 xRelPoints_F32;    /* Position relatively Left and Right point */
    tF32 alfaLeft_F32;      /* Weighting of surrounding points */
    tF32 alfaRight_F32;     /* Weighting of surrounding points */

    /* Find location (ct_xRight_S32) on axis_x where valX_F32 is located */
    while ((ct_xRight_S32 < lenX_S32) && (valX_F32 > axis_aF32[ct_xRight_S32]))
    {
        ct_xRight_S32++;
    }

    /* Index value to the left and the right of valX_F32. */
    ct_xLeft_S32  = SATURATE((ct_xRight_S32 - 1), 0, (lenX_S32 - 1));
    ct_xRight_S32 = SATURATE(ct_xRight_S32,       0, (lenX_S32 - 1));

    /* Now ct_xRight_S32 is the integer value such that axis_x[ct_xRight_S32] */
    /* is the closest axis value to the right of valX_F32, and */
    /* ct_xLeft_S32 to the left of valX_F32. If valX_F32 is outside the axis, */
    /* ct_xLeft_S32 and ct_xRight_S32 are set to the closest index in axis, */
    /* i.e. the first or the last */

    /* calculate the relative distance from the left point in axis_x */
    xdiff_F32 = axis_aF32[ct_xRight_S32] - axis_aF32[ct_xLeft_S32];
    if (xdiff_F32 > (tF32)0)
    {
        xRelPoints_F32 = (valX_F32 - axis_aF32[ct_xLeft_S32]) / xdiff_F32;
    }
    else
    {
        xRelPoints_F32 = (tF32)0;
    }

    /* Weights on surrounding points */
    alfaLeft_F32  = ((tF32)1 - xRelPoints_F32);
    alfaRight_F32 =            xRelPoints_F32;

    /* Interpolate */
    return ((vector_aF32[ct_xLeft_S32]  * alfaLeft_F32)   +
            (vector_aF32[ct_xRight_S32] * alfaRight_F32));
}
```

```
/*@ predicate is_finite_flt(real x) = -0x1.fffffep+127 <= x <= 0x1.fffffep+127;
  @ axiomatic Floor {
     logic integer floor (real x);
     axiom floor_prop: \forall real x; floor(x) <= x < floor(x)+1;
    }
  @ logic real ulp_f(real x) =
        \let e = 1 + floor(\log(\abs(x)) / \log(2));
        \pow(2,\max(e-24,-149));
  @ logic real real_ipol(real a, real b, real t) =
        (1-t)*a + t*b;
  @ logic real float_ipol(float a, float b, float t) =
        (float)(
            (float)(
                (float)(1-t)*a
            ) + (float)(t*b));
  @ lemma sign_order_invariance_real:
        \forall float a, float b, float t;
        real_ipol(a,b,t) == - real_ipol(-b,-a,(1-t));
  @ lemma sign_order_invariance_float:
        \forall float a, float b, float t;
        ( a <= b &&
          0 <= t <= 1 &&
          is_finite_flt(a) &&
          is_finite_flt(b) &&
          b-a <= 32767
        ) ==> (
          \let neg_b = (float)(-b);
          \let neg_a = (float)(-a);
          \let inv_t = (float)(1-t);
          float_ipol(a,b,t) == - float_ipol(neg_b,neg_a,inv_t)
        );
*/
/*@ requires 0 <= t <= 1;
  @ requires a <= b && b - a <= 32767;
  @ requires is_finite_flt(a) && is_finite_flt(b);
  @ ensures endpoint_ipol:
        (t == 1.f ==> \result == b) &&
        (t == 0.f ==> \result == a);
  @ ensures bounded_ipol:
        a <= \result <= b;
  @ ensures no_overflow:
        is_finite_flt(\result);
  @ ensures bounded_error:
        \abs(real_ipol(a,b,t) - \result) <= 10*ulp_f(real_ipol(a,b,t));
  @ assigns \nothing;
 */
float ipol(float a, float b, float t) {
    float t2 = 1.0f - t;
    //@ assert t == 0.f ==> t2 == 1.f;
    //@ assert t == 1.f ==> t2 == 0.f;
    float left = a*t2;
    float right = b*t;
    float sum = left+right;
    //@ assert sum == float_ipol(a,b,t);
    return sum;
}
```

## Coq proof attempt for $(1/1 + 1/0 = 1/1)$

```
(* ------------------------------------------------------------ *)
(* --- Assertion (file endpoints.c, line 14)              --- *)
(* ------------------------------------------------------------ *)
Require Import ZArith.
Require Import Reals.
Require Import BuiltIn.
Require Import int.Int.
Require Import int.Abs.
Require Import int.ComputerDivision.
Require Import real.Real.
Require Import real.RealInfix.
Require Import real.FromInt.
Require Import map.Map.
Require Import Qedlib.
Require Import Qed.
Require Import BuiltIn.
Require Import int.Int.
Require Import int.Abs.
Require Import int.ComputerDivision.
Require Import real.Real.
Require Import real.RealInfix.
Require Import real.FromInt.
Require Import map.Map.
Require Import Qedlib.
Require Import Qed.
Require Import Cmath.
Require Import Cfloat.

Require Import Axiomatic.

Theorem inv_identity: Rinv 1 = 1%R.
Proof. firstorder. Qed.
Theorem mult_identity: forall r:R, Rmult r 1 = r.
Proof. firstorder. Qed.
Theorem add_identity: forall r:R, Rplus r 0 = r.
Proof. firstorder. Qed.

Goal
  forall (r_1 r : R),
  ((r_1 <= r)%R) ->
  ((r <= (( 32767 / 1 )%R + r_1))%R) ->
  ((is_float32 r%R)) ->
  ((is_float32 r_1%R)) ->
  ((P_is_finite_flt r%R)) ->
  ((P_is_finite_flt r_1%R)) ->
  ((((add_float32 ( 1 / 1 )%R%R ( 0 / 1 )%R%R)) = ( 1 / 1 )%R)%R).

Proof.
  intros r1 r2 H1 H2 H3 H4 H5 H6. unfold add_float32.
  pose (inv_identity) as INV. unfold Rdiv. rewrite INV.
  pose (mult_identity 1) as MULT1. rewrite MULT1.
  pose (mult_identity 0) as MULT0. rewrite MULT0.
  pose (add_identity 1) as ADD1. rewrite ADD1.
(* We still need to prove (to_float32 1 = 1%R) *)
Qed.
```

### Gappa script for Endpoint Interpolation

```
@rnd = float<ieee_32,ne>;

a         = rnd(aa);
b         = rnd(bb);
t         = rnd(tt);
t2     rnd= 1-t;
left   rnd= a*t2;
right  rnd= b*t;
sum    rnd= left + right;
real      = a*(1-t) + b*t;
error     = sum - real;
rel_err   = error / real;

{
##############################################
############## absolute error ##############
##############################################
#aa in [-0x1.fffffep+127,0x1.fffffep+127] /\
#bb in [-0x1.fffffep+127,0x1.fffffep+127] /\
##############################################
############## relative error ##############
##############################################
aa in [0x1p-149,0x1.fffffep+127] /\
bb in [-0x1.fffffep+127,0x1.fffffep+127] /\
##############################################

################
### t=0 case ###
################
tt in [0,0] ->
################
### t=1 case ###
################
#tt in [1,1] ->
################

    sum in ? /\
    error in ? /\
    rel_err in ?
}
```

### Gappa script for Bounded Interpolation

```
@rnd = float<ieee_32,ne>;

a         = rnd(aa);
b         = rnd(bb);
t         = rnd(tt);
t2     rnd= 1-t;
left   rnd= a*t2;
right  rnd= b*t;
sum    rnd= left + right;
real      = a*(1-t) + b*t;
error     = sum - real;
rel_err   = error / real;

{
###############################################
############### small bounds ###############
###############################################
#aa in [-100,100] /\ bb in [-100,100] /\
###############################################
############### large bounds ###############
###############################################
aa in [-0x1.fffffep+127,0x1.fffffep+127] /\
bb in [-0x1.fffffep+127,0x1.fffffep+127] /\
###############################################

# here we state b >= a because this is guaranteed in ipol
# this is equivalent to .../\ (b-a >= 0 \/ a-b >= 0) /\...
tt in [0,1] /\ b-a >= 0 /\ b-a in [0,32767] ->

    sum in ? /\
    real in ? /\
    error in ?
}

##### Naïve Interval Arithmetic #####
# the perfect result is [min(a,b),max(a,b)]
# Gappa doesn't realize that large a(1-t) or b(t) makes the other small
# every time we reduce t's interval size in two, the error decreases
#$ t in 2;
#$ t in 4;
#$ t in 8;
#$ t in 2048;

##### Case Split + Alternate Formula #####
# if we assume that b >= a, we are subtracting a quantity of the same sign
real -> a - t*(a-b);
# and vice versa
real -> b - (1-t)*(b-a);
```

### Coq script for Jessie, sign/order invariance

```
(* Unproven Axiom! *)
Axiom Ropp_round: forall r:R,
  value (round_logic NearestTiesToEven (Ropp r)) =
  Ropp (value (round_logic NearestTiesToEven r)).

(* Why3 goal *)
Theorem sign_order_invariance_float : forall (a_2:single),
  forall (b_2:single), forall (t_2:single),
  (((value a_2) <= (value b_2))%R /\ ((0%R <= (value t_2))%R /\
  (((value t_2) <= 1%R)%R /\ ((is_finite_flt (value a_2)) /\ ((is_finite_flt
  (value b_2)) /\ (((value b_2) - (value a_2))%R <= 32767%R)%R))))) ->
  ((float_ipol a_2 b_2 t_2) = (-(float_ipol (round_logic NearestTiesToEven
  (-(value b_2))%R) (round_logic NearestTiesToEven (-(value a_2))%R)
  (round_logic NearestTiesToEven (1%R - (value t_2))%R)))%R).
Proof.
  intros a_2 b_2 t_2 (h1,(h2,(h3,(h4,(h5,h6))))).
  unfold float_ipol. rewrite Rplus_0_l. rewrite Rplus_0_l. rewrite Ropp_round.
  rewrite Ropp_round. rewrite <- Ropp_mult_distr_r. rewrite <- Ropp_mult_distr_r.
  rewrite Ropp_round. rewrite Ropp_round. rewrite <- Ropp_plus_distr.
  rewrite Ropp_round. rewrite Ropp_involutive.
  rewrite (Round_logic_def NearestTiesToEven (value b_2) (Bounded_real_no_overflow
      NearestTiesToEven (value b_2) (Bounded_value b_2))).
  rewrite (Round_logic_def NearestTiesToEven (value a_2) (Bounded_real_no_overflow
      NearestTiesToEven (value a_2) (Bounded_value a_2))).
  rewrite (Round_value NearestTiesToEven a_2).
(* Unable to prove that rnd(1-rnd(1-t2))=t2 because it's not always true *)
Qed.
```

### Gappa script for Bounding Overflow (almost)

```
@rnd = float<ieee_32,ne>;

a = rnd(aa);
b = rnd(bb);
t = rnd(tt);
t2     rnd= 1-t;
left   rnd= a*t2;
right  rnd= b*t;
result rnd= left + right;
real      = a*(1-t) + b*t;
error     = result - real;

{
# the range of a and b is one ulp smaller than +/- FLT_MAX
    aa in [-0x1.fffffcp+127,0x1.fffffcp+127] /\
    bb in [-0x1.fffffcp+127,0x1.fffffcp+127] /\
    tt in [0,1] /\ (b-a >= 32767)

->

# the result becomes exactly FLT_MAX due to rounding error
    left + right in [-0x1.fffffep+127,0x1.fffffep+127]
    /\ result in [-0x1.fffffep+127,0x1.fffffep+127]
}

# rewrite hints required in addition to reducing the Echange threshold
real -> a - t*(a-b);
real -> b - (1-t)*(b-a);
```

```
user>>> gappa -Echange-threshold=0 overflowScript.g
```

```
{
    aa in [-0x1p+39,0x1p+39] /\
    bb in [-0x1p+39,0x1p+39] /\
    tt in [0,1] /\
    b-a in [0,32767]

->

    result in [-0x1.000002p+39,0x1.000002p+39] /\
    real in [-0x1p+39,0x1p+39]
}
```