Adding Floating-point Arithmetic Support to TriCera

Danyal Mirza
Abstract

Floating-point arithmetic is a widely utilized technique for approximating real numbers. When applied in safety-critical systems, it is essential to ensure that these numbers behave as they should and do not give unsuspected errors. To ensure this, the utilization of floating-point verification is needed.

In this thesis, encoding floating-point arithmetic using the theory of rationals is explored. An algorithm that converts floating-point numbers into fractions and an algebraic data type has been implemented. Although the approximation of floating-point numbers by rationals is, in theory, neither sound nor complete, verifying the correctness of programs in the idealized setting of rationals is still very useful to increase confidence in the correctness of the program, or detect bugs. A set of basic C programs using floating-point arithmetic can now be verified and with few additions as further implementation, this method can become more sound and complete.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.2</td>
<td>Conversion of Data Types</td>
<td>24</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Long Double to Fraction</td>
<td>24</td>
</tr>
<tr>
<td>6.1.4</td>
<td>SV-COMP Benchmarks</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>7 Conclusion</td>
<td>26</td>
</tr>
<tr>
<td>A</td>
<td>Industrial Battery Example</td>
<td>29</td>
</tr>
<tr>
<td>A.1</td>
<td>The Program Code</td>
<td>29</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

On the 4th of June 1996, the launch of the rocket Ariane 5 failed. 360 million US dollars were wasted and the reason for this was a floating-point overflow when a 64-bit floating-point was converted to a 16-bit signed integer [1]. In 1994 a bug was discovered in Intel’s Pentium processor, in some cases when dividing numbers the processors would return wrong floating-point numbers. Intel was forced to recall all defective processors making this a costly mistake [2].

Bugs are unavoidable when developing software. Their severity may vary but when development is finished the goal is to have an absence of the most severe bugs that may break the system. Testing can be used to show the presence of bugs, although fails to prove that none exists. When developing critical systems, such as the previously mentioned examples, the absence of bugs has to be proven and for this program verification is needed.

Formal verification is a method for proving the correctness of a program regarding some specification. It uses methods from mathematical logic to prove the correctness and when verifying a program the code is represented as logical formulae. Formal verification is mostly used in software and hardware where safety has to be guaranteed. The C compiler CompCert [3] is an example of software that has been fully verified. This compiler can be used in critical systems and because it is formally verified it is ensured that the compiler will not make any errors.

TRICERA is an automated verification tool for C programs. It makes use of constrained Horn clauses (CHCs) and given some program and some specification TRICERA translates the program and the specification into CHCs, for which TRICERA’s back-end EL DARICA checks whether they are satisfied or not.
Currently TriCera does not have support for floating-point arithmetic. The C language (C11 standard [4]) supports three different floating-point data types: float, double and long double. In this thesis, support for these data types and operations between them will be added. The special values NaN, −∞ and +∞ will also be included. The floating-point numbers will be encoded using the theory of rationals and this thesis will explore the advantages and disadvantages of this approach.

1.1 Goals

In this thesis, floating-point arithmetic is added to TriCera and the method of encoding floating-point numbers as rationals is explored. This is an approximation but is a simple method to encode floating-point arithmetic and has some benefits which will be discussed. The encoding will be evaluated using self-written tests.

1.2 Motivating Example

Listing 1.1 shows a simplified version of industrial code from Scania, which is used for carrying out battery diagnostic checks in trucks [5]. Since the code in this example is used in vehicles, safety is important. Problems with the batteries whilst the vehicle is in operation might have devastating consequences. Therefore, using verification and making sure that the code is correct ensures some safety. Appendix A.1 shows the complete example code which has been modified to make use of floats and verifies that the voltage outputs of the battery modules are in the interval of the initialized battery voltages. Before modification, the code used integers and verified the absence of faulty battery modules.
void batteryDiag ()
{
    // Initializing the battery values
    batt_max_output = 3000.25f;
    batt_min_output = 300.25f;
    // Run the diagnostics, one module at a time
    moduleDiag(0);
    moduleDiag(1);
}

void main () {
    mod0_min = 435.75f;
    mod1_min = 500.5f;
    mod0_max = 2000.25f;
    mod1_max = 1000.42f;
    int N = _;
    assume(N > 0);
    // Run the diagnostics N amount of times
    for (int i = 0; i < N; i++) {
        batteryDiag();
    }
    assert(batt_max_output <= 3000.25f);
    assert(batt_min_output >= 300.25f);
}

Figure 1.1: The main and batteryDiag() function from the modified program module

Figure 1.1 shows the main and batteryDiag() function of the modified program module from A.1. The main function starts by initializing the values for the battery modules, giving each module a minimum and a maximum voltage. Then, at line 22 in the for-loop, batteryDiag() is called N amount of times. Line 18 sets and initializes N to _ which represents a non-deterministic value, meaning that it can be any value that is possible for an integer but with assume(N > 0) it can be any value that is larger than 0. In batteryDiag() the maximum and minimum voltage of the whole battery is set to the variables batt_max_output and batt_min_output. After all these initializations, moduleDiag(idx) run the diagnostics; one module at a time.
void modMin(int idx) {
    if (idx == 0) {
        return_modMin = mod0_min;
    } else {
        return_modMin = mod1_min;
    }
}

void modMax(int idx) {
    if (idx == 0) {
        return_modMax = mod0_max;
    } else {
        return_modMax = mod1_max;
    }
}

void moduleDiag(int idx) {
    modMin(idx);
    modMax(idx);
    //Update the max value
    if (return_modMax > batt_max_output) {
        batt_max_output = return_modMax;
    } else {
        batt_max_output = batt_max_output;
    }
    //Update the min value
    if (return_modMin < batt_min_output) {
        batt_min_output = return_modMin;
    } else {
        batt_min_output = batt_min_output;
    }
}

Figure 1.2: Three functions from the modified program module that run the diagnostics checks

The function moduleDiag(idx) shown in Figure 1.2 first returns the minimum and maximum battery voltage of the module that is currently being operated on using the functions modMin(idx) and modMax(idx). The returned voltages are compared to the minimum and maximum voltages of the entire battery and if any of these voltages exceed the interval of voltages in the entire battery, batt_max_output or batt_min_output is replaced with the voltage from the battery module.

The last two lines in line 1.2 assert (batt_max_output <= 3000.25) and assert (batt_min_output >= 300.25) try to verify that the initialized values for batt_max_output and batt_min_output remain unchanged.
This illustrates a real-life example of when verification might be used. Verification is mostly used in software where safety has to be guaranteed which is the case in this example. After all additions made in this thesis, TriCERA can verify this code.
Chapter 2

Background

2.1 TriCera

TriCera is a model checker for C programs that translates programs into constrained Horn clauses (CHCs), which TriCera’s back-end CHC solver Eldarica [6] attempts to solve. TriCera supports a large subset of the C language; however, support for floating-point arithmetic has not been implemented yet [7].

TriCera utilizes assert statements which are used to verify properties. If an assert statement fails, TriCera returns UNSAFE. If all readable assert statements in the program succeed, TriCera returns SAFE.

2.2 Formal verification in TriCera

Formal verification is a technique that verifies the correctness of a program with regards to some specification using methods from mathematical logic. It is often used in safety-critical systems where the absence of bugs is important. There exist several different techniques in formal verification. One of them is model checking which as mentioned before, TriCera makes use of.

2.2.1 Sound, Complete and Terminating

When creating verification software, three important properties a tool can have are: soundness, completeness and termination. Soundness ensures that the tool only produces correct results and completeness ensures that every result that is in fact true can be proved [8][pp. 39] and termination ensures that the tool terminates and a result is received. In the context of TriCera soundness means that all solutions that are modeled safe are in fact safe and can not be unsafe. Completeness means that all solutions modeled unsafe are in fact unsafe and can not be safe. All these properties are desired but
when creating verification tools often all of these cannot be satisfied simultaneously. This is because of the incompleteness theorem which says that true properties exist that can not be proven to be true [9]. Some properties may also be too complex such that a tool that tries to verify such property may not terminate, making such a property non-terminating, and some complex properties that will terminate might not terminate within a reasonable amount of time, also causing problems.

2.3 Floating-point Arithmetic

Floating-point arithmetic is a widely used way to approximately represent real numbers. They are represented with a significand, a sign-bit and an exponent. An example of this is:

\[ 2.625 = -1^0 \cdot 1.0101 \cdot 2^1 \]

In this example, 0 in \(-1^0\) is the sign-bit, 0101 is the significand and 1 in \(2^1\) is the exponent [10].

2.3.1 IEEE-754

IEEE-754 is the standard for floating-point arithmetic and provides information about how floating-point numbers should be implemented. This standard contains information about which values a float can be, how operations should be implemented and the attributes of rounding [11].

Values

Zeroes +0 and −0 exist, they are equal to each other and behave equally. The difference between them is that their sign bit is either 1 or 0.

Infinity +∞ and −∞ exist. They are represented by an exponent of all ones and a significand of all zeroes. What differentiates them is their sign bit.

Not a Number (NaN) A numeric data type that is interpreted as undefined, there exist two different NaNs: Signaling NaNs and quiet NaNs. Signaling NaNs are represented by an exponent of all zeroes and a significand with a leading zero, they occur when performing operations that are not valid. Quiet NaNs are represented by an exponent of all zeroes and a significand with a leading one, and they represent an intermediate value. An example of this occurring is the result of dividing by ∞ or multiplying ∞ with zero.
Directed Rounding

round-to-$+\infty$ The floating-point number should be rounded to the number closest to, but not less than the precise number.

round-to-$-\infty$ The floating-point number should be rounded to the number closest to, but not greater than the precise number.

round-to-0 The floating-point number should be rounded to the number closest to, but not greater in magnitude than the precise number.

Round-to-nearest

round-ties-to-even The floating-point number should be rounded to the number nearest to the precise number. If two numbers are equally near, the number with an even least significant bit should be returned. If that is not possible, the one with the largest magnitude should be returned.

round-ties-to-away The floating-point number should be rounded to the number nearest to the precise number. If two numbers are equally near, the number with the largest magnitude should be returned.

2.3.2 Floating-point Arithmetic in C

The C language has three data types for floating-point numbers, float, double and long double. Table 2.1 shows the precision of the different data types and the size of the significand, exponent and sign-bit. According to the IEEE-754 [11] standard the precision of a long double is at least as big as a double, the GNU C compiler (GCC) [12] sets the precision of long double to 80 bits.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Precision</th>
<th>Sign bit</th>
<th>Exponent</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32 bits</td>
<td>1 bit</td>
<td>8 bits</td>
<td>23 bits</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td>1 bit</td>
<td>11 bits</td>
<td>52 bits</td>
</tr>
<tr>
<td>long double</td>
<td>80 bits</td>
<td>1 bit</td>
<td>15 bits</td>
<td>64 bits</td>
</tr>
</tbody>
</table>
Complying to the IEEE-754 standard NaNs, $+\infty$ and $-\infty$ do exist as well. Tables 2.2 and 2.3 show which operations that cause these special values.

Table 2.2: Operations that cause NaN, $w \in \mathbb{R}$

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0/0$</td>
<td>$-\infty$ if $w &lt; 0$</td>
</tr>
<tr>
<td>$\pm\infty/\pm\infty$</td>
<td>$-\infty$ if $w &lt; 0$</td>
</tr>
<tr>
<td>$0 \cdot \pm\infty$</td>
<td>$-\infty$ if $w &lt; 0$</td>
</tr>
<tr>
<td>$\pm\infty \cdot 0$</td>
<td>$-\infty$ if $w &lt; 0$</td>
</tr>
<tr>
<td>$+\infty - \infty$</td>
<td>$-\infty$</td>
</tr>
</tbody>
</table>

Any operation involving NaN

Table 2.3: Operations that cause $\infty$, $w \in \mathbb{R}$

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w/ + 0$</td>
<td>$+\infty$ if $w &gt; 0$</td>
</tr>
<tr>
<td></td>
<td>$-\infty$ if $w &lt; 0$</td>
</tr>
<tr>
<td>$w/ - 0$</td>
<td>$+\infty$ if $w &lt; 0$</td>
</tr>
<tr>
<td></td>
<td>$-\infty$ if $w &gt; 0$</td>
</tr>
<tr>
<td>$0^{-1}$</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>$-0^{-1}$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>$+\infty + \infty$</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>$-\infty - \infty$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>$+\infty + w$</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>$+\infty - w$</td>
<td>$-\infty$ if $w &lt; 0$</td>
</tr>
<tr>
<td>$-\infty + w$</td>
<td>$+\infty$ if $w &gt; 0$</td>
</tr>
<tr>
<td>$-\infty - w$</td>
<td>$-\infty$ if $w &lt; 0$</td>
</tr>
<tr>
<td>$-\infty / w$</td>
<td>$+\infty$ if $w &gt; 0$</td>
</tr>
<tr>
<td></td>
<td>$-\infty$ if $w &lt; 0$</td>
</tr>
<tr>
<td>$+\infty / w$</td>
<td>$+\infty$ if $w &gt; 0$</td>
</tr>
<tr>
<td></td>
<td>$-\infty$ if $w &lt; 0$</td>
</tr>
<tr>
<td>$-\infty / w$</td>
<td>$+\infty$ if $w &lt; 0$</td>
</tr>
<tr>
<td></td>
<td>$-\infty$ if $w &gt; 0$</td>
</tr>
</tbody>
</table>

When doing an operation of numbers with different data types, a conversion has to be made to a specific data type. Generally in C, the values are converted to the data type with larger precision. Table 2.4 gives some examples of how floating-point numbers are converted with non-matching data types.

Table 2.4: A non-exhaustive list of operations between non-matching data types

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>int op float</td>
<td>float</td>
</tr>
<tr>
<td>float op double</td>
<td>double</td>
</tr>
<tr>
<td>double op long double</td>
<td>long double</td>
</tr>
</tbody>
</table>

2.3.3 Difficulties Verifying Floating-point numbers

Obtaining sound analysis for floating-point arithmetic can be difficult. There exist both architectural and mathematical faults that make analysis more complex. Computing transcendental functions (e.g. $\sin$, $\cos$ and $\log$) may give different results depending on what processor is used.
The default rounding for the C language (C11 standard) is round-to-nearest [4]. In [13], it is discovered that there exist floating-point libraries that do not comply with this standard and have the default rounding mode set to something else. It is also mentioned that for some floating-point libraries, the rounding modes that are not round-to-nearest are poorly tested and do not work as intended.

2.4 Rationals

Rationals ($\mathbb{Q}$) is a set of numbers that can be expressed as a fraction of two integers with a non-zero denominator. An example of a rational number is $0.25$ which can be written as $1/4$. Numbers that cannot be written as a fraction of two integers exist, such as $\pi$, $e$ and $\sqrt{2}$. These numbers are irrational and are not part of the rational set.

2.5 Algebraic Data Type

An algebraic data type (ADT) is a data type that is composed by combining several types, these types can hold different values and can also be recursive which means that one of the combined types in the ADT can be the ADT itself. Figure 2.1 is an ADT that represents a linked list written in the functional programming language Haskell. The linked list has two constructs: Empty, which represents an empty node in the list and Cons a (LinkedList a), which represents a non-empty node in the list where Cons a contains the data and LinkedList a is a reference to the next node in the list [14].

```
1   data LinkedList a = Empty
2      | Cons a (LinkedList a)
```

Figure 2.1: An ADT of a linked list

2.6 Constrained Horn Clauses

A clause is a disjunction of literals. A Horn clause is a clause with at most one positive literal. An example of a Horn clause is $\neg P_1 \lor \neg P_2 \lor \ldots \lor \neg P_n \lor U$ which is written as a disjunction, where $U$ is a positive literal and the rest are negative literals. Horn clauses are often written as implications, the previous clause can be written as $U \leftarrow P_1 \land P_2 \ldots \land P_n$.

Extending Horn clauses with a constraint over some background theories we get constrained Horn clauses (CHCs). Based on the example previously
mentioned a constrained Horn clause would be $U \leftarrow P_1 \land P_2 \ldots \land P_n \land C$ where $C$ is a constraint.

Figure 2.2 is an example of how C code can be translated into CHCs [15]. The clause $s_0(0) \leftarrow true$ which corresponds to line 2 in the code is interpreted as if $true$ holds $s_0(0)$ also holds, which assigns the value 0 to a variable. This clause is satisfied when $s_0(0)$ is true. In the next clause, it is specified that the value from the previous clause is assigned to the variable $x$ and if $s_0(x)$ holds the value of $x$ is assigned to a new variable which is specified in the third clause as $y$.

The third and fourth clauses represent the execution of the while-loop and have two different constraints dependent on if the while-loop is executed or not. The third clause, with the constraint $2 \geq x$ represents entering the loop and if it is satisfied, $s_2(x, y)$ holds. The fifth and sixth clauses represent the inside of the while-loop and are only satisfied if $s_2(x, y)$ holds. In these clauses 1 is added to $x$ and $y$ is assigned the value of $x$.

The last clause represents the assert statement and gives an error state if it is satisfied. The clause specifies that $s_3(x, y)$ and $x \neq y$ must not hold at the same time. $s_3(x, y)$ only holds if the fourth clause is satisfied, which represents not entering the while-loop.

```c
int main() {
    int x = 0;
    int y = x;
    while (x <= 2) {
        x++; y = x;
    }
    assert(x==y);
}
```

Figure 2.2: C code and its represented CHCs

```c
s_0(0) \leftarrow true
s_1(x, x) \leftarrow s_0(x)

s_2(x, y) \leftarrow s_1(x, y), 2 \geq x
s_3(x, y) \leftarrow s_1(x, y), 2 < x

s_4(x + 1, y) \leftarrow s_2(x, y)
s_1(x, x) \leftarrow s_4(x, y)
false \leftarrow s_3(x, y), x \neq y
```
Chapter 3

Adding Floating-point Arithmetic to TRICERA

In this thesis floating-point numbers are encoded using the theory of rationals. For this implementation two major additions have to be made, converting floating-point numbers into fractions and implementing an ADT to allow special values. This chapter will explain the logical aspects of these additions and then discuss these aspects.

3.1 Encoding Floating-point numbers as Rationals

Rationals, as discussed in Section 2.4 are defined as fractions of two integers. When verifying floating-point numbers they first have to be converted into fractions which is done with an algorithm that is discussed in detail in Section 4.2. The logic of how floating-point numbers are represented and constructed is used in the algorithm. The significand, exponent and sign-bit are extracted and operated on to receive a denominator and numerator.

3.1.1 Encoding with the ADT

The IEEE-754 standard (Section 2.3.1) discusses how floating-point numbers should be able to hold the special values: NaN, $+\infty$ and $-\infty$. Rationals can not by default hold these values as they are not elements of $\mathbb{Q}$. To allow this support, an ADT is implemented allowing floating-point numbers to be a rational number, NaN, $+\infty$ or $-\infty$. Figure 3.1 is an illustration and a definition of the ADT written in SMT-LIB [16].
Basic arithmetic operations have been implemented using FloatADT, Table 3.1 defines the implemented arithmetic operations and Table 3.2 defines the implemented predicates for the ADT. These semantics describe for each of the implemented operations and predicates, what should occur when using the different data values in the ADT.

Deviations from the IEEE-754 Standard

These semantics are not exhaustive and deviate from the IEEE-754 standard. Negative and positive zeroes are not implemented in FloatADT, thus not allowing operations including them and the operation \( w/\infty \) does not result in zero as it should. Instead according it results in NaN. A case for multiplication and division where the semantics deviates is: \( +\infty \times w \) and \( +\infty /w \) where \( w < 0 \). Regarding the IEEE-754 standard these operations should equal \(-\infty\) but with the current semantics this equals \(+\infty\) due to lack of implementation.

The IEEE-754 standard describes more operations than defined in the current semantics. Examples of operations that are not implemented are exponentiation, square root, and conversion to different data types. Table 2.4 is a non-exhaustive list of how conversions should be handled and Table 2.3 gives an example of when exponentiation causes \( \infty \).
a + b =
FloatData(frac(a) + frac(b)) if isFloat(a) ∧ isFloat(b)

plusInf() if isPlusInf(a) ∧ isFloat(b) ∨ isFloat(a) ∧ isPlusInf(b)

negInf() if isNegInf(a) ∧ isFloat(b) ∨ isFloat(a) ∧ isNegInf(b)

NaN() otherwise

a - b =
FloatData(frac(a) - frac(b)) if isFloat(a) ∧ isFloat(b)

plusInf() if isPlusInf(a) ∧ isFloat(b) ∨ isFloat(a) ∧ isNegInf(b)

negInf() if isNegInf(a) ∧ isFloat(b) ∨ isFloat(a) ∧ isPlusInf(b)

NaN() otherwise

a * b =
FloatData(frac(a) * frac(b)) if isFloat(a) ∧ isFloat(b)

plusInf() if isPlusInf(a) ∧ isFloat(b) ∨ isFloat(a) ∧ isPlusInf(b)

negInf() if isNegInf(a) ∧ isFloat(b) ∨ isFloat(a) ∧ isNegInf(b)

NaN() otherwise

a / b =
FloatData(frac(a)/frac(b)) if isFloat(a) ∧ isFloat(b)

plusInf() if isPlusInf(a) ∧ isFloat(b)

negInf() if isNegInf(a) ∧ isFloat(b)

NaN() otherwise

Table 3.1: Semantics for implemented arithmetic operations using the ADT
a > b = 
FloatData(frac(a) > frac(b)) if isFloat(a) ∧ isFloat(b)
True if isPlusInf(a) ∧ isFloat(b) ∨ isFloat(a) ∧ isNegInf(b)
False otherwise

otherwise

a ≥ b = 
FloatData(frac(a) ≥ frac(b)) if isFloat(a) ∧ isFloat(b)
True if isPlusInf(a) ∧ isFloat(b) ∨ isFloat(a) ∧ isNegInf(b)
False otherwise

a < b = 
FloatData(frac(a) < frac(b)) if isFloat(a) ∧ isFloat(b)
True if isNegInf(a) ∧ isFloat(b) ∨ isFloat(a) ∧ isPlusInf(b)
False otherwise

a ≤ b = 
FloatData(frac(a) ≤ frac(b)) if isFloat(a) ∧ isFloat(b)
True if isNegInf(a) ∧ isFloat(b) ∨ isFloat(a) ∧ isPlusInf(b)
False otherwise

a == b = 
FloatData(frac(a) == frac(b)) if isFloat(a) ∧ isFloat(b)
True if isPlusInf(a) ∧ isPlusInf(b) ∨ isNegInf(a) ∧ isNegInf(b)
False otherwise

a ≠ b = 
FloatData(frac(a) ≠ frac(b)) if isFloat(a) ∧ isFloat(b)
False if isPlusInf(a) ∧ isPlusInf(b) ∨ isNegInf(a) ∧ isNegInf(b)
True otherwise

Table 3.2: Semantics for predicates using the ADT

3.1.2 Generating CHCs

The additions of the FloatADT and the algorithm that converts floating-point numbers to fractions allow floating-point numbers to be represented in CHCs. Figure 3.2 shows the CHCs of a simple program that asserts that the sum of 0.5 and 0.75 is 1.25. The first clause assigns fractional values to some variables, floatData is an element of FloatADT used to encapsulate fractions. The second clause checks if 0.5 + 0.75 = 1.25. getFloat(a) and getFloat(b) unwraps the values, thereby allowing the addition. To be able to compare the result from the addition with 1.25, floatData wraps up the addition of a and b into a single value. getFloat unwraps the values to allow the disequality operation between floatData(getFloat(a) + getFloat(b)) and floatData(Rat_frac(5,4)). Rat_frac() is used to rep-
resent rational values as fractions. It takes in two integers and wraps the
values to represent a rational number.

\[ s_0(a, \text{floatData}(\text{Rat}_\text{frac}(3, 4))) \leftarrow a = \text{floatData}(\text{Rat}_\text{frac}(1, 2)). \]
\[ \text{false} \leftarrow s_0(a, b), \text{getFloat}(\text{floatData}(\text{getFloat}(a) + \text{getFloat}(b))) \]
\[ \neq \text{getFloat}(\text{floatData}(\text{Rat}_\text{frac}(5, 4))) \]

Figure 3.2: CHCs of a program that asserts that the sum of 0.5 and 0.75 is
1.25

### 3.2 Limitations

Verifying floating-point numbers in a sound and complete manner is a difficul-
task. As discussed in Section 2.3.3, there are many factors that make it
troublesome to obtain sound analysis. The method used in this thesis is not
sound or complete. As rationals are mathematical and not a representation
of how floating-point numbers are modeled in the computer, rounding-errors
and overflows are not modeled correctly. Further limitations of the used
method, which will be explored in greater depth, include the inability to
model rounding modes and the approximations resulting from converting
floating-point numbers into fractions.

Since the floating-point numbers are encoded as rationals, the different round-
ing modes mentioned in Section 2.3.1 are not considered. This makes the
method deviate from the IEEE-754 standard and may prevent sound and
complete analysis. However, as discussed in Section 2.3.3 rounding modes
in floating-point libraries may be set to something that is not default and
some of the rounding modes may be poorly tested. Avoiding these potential
errors gives better confidence that the verification is correct.

In the C language, round-ties-to-even is the default rounding mode. In
[13] and as discussed in Section 2.3.3, it would be naive to think that all
floating-point libraries handle this correctly for all cases and choose always
round-ties-to-even as default. If a program that does not use the default
floating-point library needs to be verified all of this needs to be taken into
account, making sure rounding for all of the different modes works correctly
and the default is what it should. This whole process is time-consuming and
adds complexity to the verification process.

The method used for converting floating-point numbers into fractions in this
thesis gives approximations, these are neither under-approximation nor
over-approximation. The method of converting floating-point numbers into fractions which, will be discussed in Section 4.2 produces approximations due to the numerator and denominator in the produced fraction only being able to be an exponent with a base that is 2. A fraction will always be in the form $2^n/2^m$ where $n$ and $m \in \mathbb{Z}_0^+$.

```c
int main() {
    float a = 4.2f;
    float b = 0.3f;
    assert (a/b == 14.0f);
}
```

Figure 3.3: Example program that will give approximation errors

Figure 3.3 shows an example of when these approximation errors may occur. 4.2/0.3 should be equal to 14.0 but trying to verify this program returns unsafe. Converting both floating-point numbers into fractions gives $a = \frac{17616076}{4194304}$ and $b = \frac{5033165}{16777216}$. The numbers 4.2 and 0.3 are not represented precisely since the denominators of 42/10 and 3/10 cannot be written as an exponentiation where the base is 2, thus being approximated to something as close as possible to 42/10 and 3/10 where the denominators are an exponentiation with the base 2. Dividing $a$ and $b$ gives the result of $13.99999880790715186208$, which is not equal to 14.0. This example also fails in the C language using GCC with default rounding modes. Consequently, when asserting a floating-point value, it is advisable to employ interval checks instead of exact equality tests. The assert statement for this example should instead be rewritten as: `assert((a/b) >= 14.0f - \epsilon && a/b <= 14.0f + \epsilon)` where $\epsilon$ is some specified value to take the imprecision into consideration. Due to this approximation, floating-point values should always be verified using intervals. With this simple example, the margin of error is quite small. However, it should be noted that the error grows larger if the operation is repeated multiple times. When creating the assert statement and specifying $\epsilon$, this should be taken into consideration.

Verification is mainly used in critical systems such as cars and industrial equipment. In these examples, the program to verify often uses values from sensors that might produce noisy data. It could be argued that the produced approximation from converting floating-point numbers into fractions is insignificant when dealing with noisy data since approximating something already approximate might bring it closer to the real result in some cases. However, this cannot be assumed since there are situations where this approach could lead to even worse outcomes.
Chapter 4

Implementation

TriCera is a verifier for the C language, thus the implementation has to be in respect of how floating-point numbers are implemented in C and the existing data types as mentioned in Section 2.3.2. IEEE-754 standard (Section 2.3.1) mentions rounding and the different rounding modes. In this implementation, this does not have to be taken into account because the theory of rationals is used to encode the floating-point numbers. The data types that should be implemented are: float, double and long double and the special values NaN, +∞ and −∞ should also be implemented.

4.1 Parsing Floating-point numbers

The syntax for floating-point numbers is written in Backus-Naur-Form (BNF) [17].

For parsing, the BNFC tool [18] is used and produces an abstract syntax tree that separates the different parts of the program. The different floating-point numbers are represented in the same way as they are represented in the C language. Figure 4.1 shows the BNF grammar for floating-point numbers.

(\langle CFloat \rangle ::= (((\langle digit \rangle + '.) (\langle digit \rangle)+ ) | (\langle digit \rangle+ '.) | ('.\langle digit \rangle+)(('e'|'E') ('-')? (\langle digit \rangle)+)?('f'|'F'))) |

(\langle CDouble \rangle ::= (((\langle digit \rangle+ '.) (\langle digit \rangle)+ ) | (\langle digit \rangle+ '.) | ('.\langle digit \rangle+)(('e'|'E') ('-')? (\langle digit \rangle)+)?((\langle digit \rangle+ ('e'|'E')('-')? (\langle digit \rangle+))('f'|'F'))) |

(\langle CLongDouble \rangle ::= (((\langle digit \rangle+ '.) (\langle digit \rangle)+ ) | (\langle digit \rangle+ '.) | ('.\langle digit \rangle+)(('e'|'E') ('-')? (\langle digit \rangle)+)?('l'|'L'))) |

Figure 4.1: BNF grammar for floating-point numbers in C
4.2 Converting Floating-point numbers into Fractions

As we represent floating-point numbers as rationals, each floating-point number has to be converted into a fraction of two integers. This is done with an algorithm that first converts the floating-point number into binary and extracts the significand, exponent and sign-bit. The theory of this algorithm is based on the definition of a floating-point number explained in Section 2.3.

A floating-point number consists of three parts, the exponent, the significand and sign-bit which are multiplied by each other to create a floating-point number. The exponent represents an exponentiation with a base of 2, the significand represents a rational number that is larger or equal to 1 and smaller or equal to 2 and the sign-bit is an exponentiation with a base of $-1$ and an exponent of either 0 or 1 depending on if the number is negative or not. In the following algorithms, the exponent and significand are binary numbers of the data-type string thus allowing iteration to each bit. Each bit in the significand is represented with the formula $2^{-n}$ where $n$ is the position of that bit and the full number of the significand is the sum of all 1’s. This theory is used in the following algorithms to achieve a fraction since $2^{-n} = 1/2^n$.

In the following algorithms, there are two undefined auxiliary functions: \texttt{len()} and \texttt{reverse()}. \texttt{len()} takes a string as input and outputs the total amount of characters in that string, \texttt{reverse()} takes a string as input and outputs it reversed. As seen in Algorithm 1, the first for-loop iterates through the reversed significand and acquires the denominator. The second for-loop acquires the numerator by iterating through the significand and adding $\text{denominator}/2^{\text{bitCount}}$. If the exponent and the sign-bit are zero, the result obtained from this is the used fraction. If the exponent is not zero it has to be calculated using Algorithm 2 and then multiplied with either the numerator or denominator depending on if the exponent is less than zero or more as seen in Algorithm 3. If the sign-bit is one, the numerator is negated.
Algorithm 1 Obtaining the numerator and denominator from the significand.

Convert the floating-point number to binary and extract the significand, exponent and sign-bit.

```python
int bitCount = len(significand)
for i in reverse(significand) do
    if i == 1 then
        denominator = 2^bitCount
        break
    end if
    bitCount = bitCount - 1
end for

bitCount = 1
numerator = denominator
for i in significand do
    if i == 1 then
        numerator = numerator + denominator/2^bitCount
    end if
    bitCount = bitCount + 1
end for
```

Algorithm 2 Obtaining the exponent

```python
int bitCount = 0
exponentInt = -2^{len(exponent)} - 1 + 1
for i in exponent do
    if i == 1 then
        exponentInt = exponentInt + 2^bitCount
    end if
end for
```
Algorithm 3 Combining the result from the significand with the result from the exponent

\[
\text{if exponentInt} > 0 \text{ then} \\
\quad \text{numerator} = \text{numerator} \times 2^{\text{exponentInt}} \\
\text{end if}
\]

\[
\text{if exponentInt} < 0 \text{ then} \\
\quad \text{denominator} = \text{denominator} \times 2^{\text{exponentInt}} \\
\text{end if}
\]

\[
\text{if signBut} == 1 \text{ then} \\
\quad \text{numerator} = -\text{numerator} \\
\text{end if}
\]

\[\text{return (numerator, denominator)}\]

4.3 NaN and Infinity

To be able to represent NaN and Infinities an algebraic data type (ADT) is used. Using the ADT the floating-point values NaN, $+\infty$ and $-\infty$ as discussed in Section 2.3.1 can be represented. The logic behind the implemented ADT is discussed in detail in Section 3.1.1. For each data type an ADT has been implemented; float, double and long double. Each ADT has the implemented functions isFloat(), isNan(), isPlusInf() and isNegInf() that allow checking if a value is a part of ADT.

4.4 Unit and Regression Tests

Several regression tests have been made to test the implementation\footnote{All of the implemented benchmarks can be seen on https://github.com/dannem1337/TriCera/tree/master/regression-tests}. Most of them test simple cases for each operation but a few of them are a bit more challenging and test edge cases. For more thorough testing both SAFE and UNSAFE tests have been implemented. For almost every SAFE test, an UNSAFE version has been made.

Unit tests have been written for the algorithm in Section 4.2. These tests are made to ensure the correctness of the algorithm and catch errors if something is changed.

\footnote{All of the implemented benchmarks can be seen on https://github.com/dannem1337/TriCera/tree/master/regression-tests}
Chapter 5

Related Work

In this chapter, we will discuss some related work mainly focusing on different floating-point implementations in other verification tools. In this thesis, floating-point arithmetic is encoded with the theory of rationals and there are other tools that use similar methods.

5.1 Floating-point Implementations

The EVA plug-in \[19\] is a part of the verification tool Frama-C and has sound analysis for floating-point arithmetic. EVA represents floating-point numbers as intervals and uses interval arithmetic for operations. There is support for all of the special values mentioned by the IEEE-754 standard (Section 2.3.1), EVA emphasizes that infinities and NaNs are unwanted errors and alerts users when they may occur.

CBMC \[20\] is a model checker for C programs that supports bit-accurate reasoning about floating-point arithmetic. The floating-point numbers are mapped to IEEE-754 binaries, floats are mapped to Binary32 and doubles are mapped to Binary64. CBMC also makes use of the rounding modes described by IEEE-754 (Section 2.3.1) and uses round-to-nearest as default.

The method of using bit-accurate reasoning about floating-point numbers that is used in CBMC gives a more sound and complete analysis compared to using rationals to encode floating-point numbers. With this method, rounding errors and overflows can be modeled. However, it adds complexity for the developer as rounding modes have to be considered. Using intervals to represent floating-point numbers as used in the EVA plugin-in in Frama-C would eliminate the approximation errors produced from converting floating-point numbers into fractions.
Chapter 6

Results

A basic implementation for verifying floating-point arithmetic has been implemented. Table 6.1 shows which operations are supported, basic regression tests for these operations have been implemented for all data types and are working as intended.

Table 6.1: C operations that TriCERA support for floating-point arithmetic.

<table>
<thead>
<tr>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -, *, /, &lt;, ≤, &gt;, ≥, ==, ≠</td>
</tr>
</tbody>
</table>

Figure 6.1 shows two examples of what TriCERA now can encode and verify correctly. The left example does operations on \( x \) in a loop and assigns the values to \( y \). The assertion checks that \( x \) is equal to \( y \) and this example returns \( \text{SAFE} \) which is correct. The other example makes use of the non-deterministic integer \( N \) which can have any integer value. The while loop is iterated \( N \) times and in the first iteration where \( x == y \), \( 1.2f \) is added to \( x \). This code example returns \( \text{UNSAFE} \) as it should.

Most of the implemented regression tests work as intended. The failing tests are caused by a lack of implementation in some areas which will be discussed in Section 6.1.
6.1 Further Implementations

This thesis has implemented a base for verifying floating-point arithmetic. To get a more complete implementation supporting all the features in the C language more implementation has to be added in some areas.

6.1.1 NaN and Infinity

A semantics for arithmetic operations and predicates is described in Section 3.1.1 but since there is no way of creating a NaN or \( \infty \) they can not be verified. The current semantics does not fully comply with the IEEE-754 standard but works as a base for further implementation.

The IEEE-754 standard (Section 2.3.1) differentiates signaling and quiet NaNs. The current implementation treats all NaNs the same.

6.1.2 Conversion of Data Types

Conversion of data types and support for operations between non-matching data types is also not supported. Table 2.4 shows a non-exhaustive example of how operations between non-matching data types are handled. Since this list is non-exhaustive, research into whether any examples deviate from this pattern should be made.

6.1.3 Long Double to Fraction

The algorithm mentioned in Section 4.2 works as intended for float and double but not for long double. Currently, the algorithm extracts the sign-bit, exponent and significand the same way as double which is incorrect since long double has a larger precision of 80 bits. To convert the
floating-point number into binary, the Java functions `floatToIntBits()` and `doubleToIntBits()` have been used for the data types `float` and `double`. Since there are no 80 bit data types in Java or Scala a new function which takes a string as input and returns 80 bit floating-point number in binary.

### 6.1.4 SV-COMP Benchmarks

Currently, floating-point arithmetic for TrICERA is evaluated using self-written tests. For a better evaluation of the implementation, SV-COMP [21] benchmarks should be used which makes it easier to evaluate the method used in this thesis against other verification tools.
Chapter 7

Conclusion

In this thesis, a base for encoding floating-point arithmetic has been implemented in TriCera. The method of encoding uses the theory of rationals to represent floating-point numbers and an algebraic data type to include the special values: NaN, $+\infty$ and $-\infty$. The encoding as rationals makes use of an algorithm that converts the floating-point numbers into a fraction of two integers which is represented in the algebraic data type included with the special values. Semantics for all implemented floating-point operations has been constructed, specifying how operations are handled for each data value in the algebraic data type.

This method of encoding is not sound or complete but still shows some promising results. A set of C programs using floating-point arithmetic can now be verified and the used technique does have some benefits over other methods, such as not needing to take rounding modes into account. To get this implementation to its full potential, more work has to be done, mostly in making sure that everything follows the IEEE-754 standard which will require a few more additions.
Bibliography


Appendix A

Industrial Battery Example

The following is the complete code of the motivating example discussed in Section 1.2. The code is originally from [5] but has been modified to make use of floats.

A.1 The Program Code

```c
// Global variables 'acting' as return variables
float return_modMin;
float return_modMax;
int return_modStatus;

int mod0_status;
int mod1_status;

float mod0_min;
float mod1_min;

float mod0_max;
float mod1_max;

float batt_min_output;
float batt_max_output;
int batt_status_output;

int N = _;

void modStatus(int idx) {
    if (idx == 0) {
        return_modStatus = mod0_status;
    } else {
        return_modStatus = mod1_status;
    }
}

void modMin(int idx) {
```

```c
```
if (idx == 0) {
    return_modMin = mod0_min;
} else {
    return_modMin = mod1_min;
}

void modMax(int idx) {
    if (idx == 0) {
        return_modMax = mod0_max;
    } else {
        return_modMax = mod1_max;
    }
}

void moduleDiag(int idx) {
    modMin(idx);
    modMax(idx);
    modStatus(idx);

    // Update the status
    if (return_modStatus == 1) {
        batt_status_output = 1;
    } else {
        batt_status_output = batt_status_output;
    }

    // Update the max value
    if (return_modMax > batt_max_output) {
        batt_max_output = return_modMax;
    } else {
        batt_max_output = batt_max_output;
    }

    // Update the min value
    if (return_modMin < batt_min_output) {
        batt_min_output = return_modMin;
    } else {
        batt_min_output = batt_min_output;
    }
}

void batteryDiag() {
    // Initializing the battery values
    batt_max_output = 3000.25f;
    batt_min_output = 300.25f;
    batt_status_output = 0;

    // Run the diagnostics, one module at the time
    moduleDiag(0);
    moduleDiag(1);
}
void main()
{
    return_modMin = 0.0 f;
    return_modMax = 0.0 f;

    return_modStatus = 0;
    mod0_status = 0;
    mod1_status = 0;

    mod0_min = 435.75 f;
    mod1_min = 500.5 f;
    mod0_max = 2000.25 f;
    mod1_max = 1000.42 f;

    int i;
    assume (N > 0);

    // Run the diagnostics N amount of times
    for (i = 0; i < N; i++) {
        batteryDiag();
    }

    assert (batt_max_output <= 3000.25 f);
    assert (batt_min_output >= 300.25 f);
}