Bit-Vector Approximations of Floating-Point Arithmetic

Joel Havermark
Abstract

Bit-Vector Approximations of Floating-Point Arithmetic

Joel Havermark

The use of floating-point numbers in safety-critical applications shows a need to efficiently and automatically reason about them. One approach is to use Satisfiability modulo theories (SMT). The naive approach to using SMT does not scale well. Previous work suggests approximations as a scalable solution. Zeljic, Backeman, Wintersteiger, and Rümmer have created a framework called UppSAT for iterative approximations. The approximations created with UppSAT use a precision to indicate how approximate the formula is.

Floating-point can be approximated by the simpler fixed-point format. This provides the benefit of not having to encode the rounding modes and special values. It also enables efficient encodings of the operations as bit-vectors. Zeljic et al. have implemented such an approximation in UppSAT. The precision of the approximation indicates the amount of bits that the numbers use. This thesis aims to improve the way that the approximation handles precision: by providing two new strategies for increasing it, increasing the detail, and changing the maximum. One of the two new strategies is implemented. Both strategies are based on the idea to calculate the amount of bits needed to represent a float as fixed-point.

The implemented strategy performs worse than the current but solves test cases that the current is not able to solve. The reason for the performance is probably a too fast increase of the precision and that the same precision is used for the whole formula. Even though the implemented strategy is worse, the new strategies and precision domain can provide a base to build on when further improving the approximation and also help to show that fixed-point and approximations, in general, are suitable for reasoning about floating-point and that there are more approximations to investigate.
## Contents

1 Introduction .................................................. 2  
2 Background .................................................... 3  
  2.1 Satisfiability Modulo Theories ............................. 3  
  2.2 UppSAT .................................................. 5  
  2.3 Floating-Point Arithmetic ................................. 8  
  2.4 Fixed-Point Arithmetic ................................. 9  
3 Refinement Strategies ........................................ 9  
  3.1 Precision and Refinement ................................ 10  
  3.2 Uniform Refinement Strategy .............................. 12  
  3.3 Compositional Refinement Strategy ....................... 12  
4 Evaluation of Uniform Refinement Strategy ................. 13  
  4.1 Comparison of Maximum Precisions ....................... 14  
  4.2 Comparison with the Old Implementation ................. 15  
  4.3 Comparison with the Small-Floats Approximation ......... 16  
5 Related Work .................................................. 17  
6 Conclusion & Future Work ..................................... 19
1 Introduction

Correct programs are important and difficult to write. The difficulty of knowing if a program is correct and the prevalence of floating-point numbers in safety critical applications [14] shows a need to efficiently and automatically reason about floating-point arithmetic (FPA). One approach is to use Satisfiability modulo theories (SMT) [12].

SMT is an extension to the Boolean satisfiability problem (SAT) [2]. SMT extends SAT with background theories allowing other symbols and values than the ones from propositional logic [12]. For an explanation of SAT and SMT see Section 2.1. Brain, Tinelli, Rümmer, and Wahl have developed a background theory for FPA [4]. The current methods of solving problems expressed in the theory of FPA suffer from scalability problems [18]. Previous work proposes different kinds of approximations to deal with the scalability problem. Brillout et al. have created a framework for iteratively approximating a floating-point formula by using mixed approximations [5]. Ramachandran and Wahl propose a similar scheme for iterative approximation, by reducing the problem to another theory [17].

A third way to use approximations is the systematic approximation framework UppSAT [19]. UppSAT combines a specification of an approximation and a decision procedure for the relevant theories and creates an approximating SMT-solver. The parts of an approximation in UppSAT that are most relevant for this thesis is the precision and the refinement. The precision determines the degree of approximation and the refinement decides by how much the approximation should come closer to the original formula each iteration.

UppSAT calls the external solver with the result of applying the approximation to the original formula. There are two possibilities depending on the output from the external solver: 1. UppSAT terminates and returns the model, if the model for the approximated formula is a model for the original formula; 2. UppSAT changes the degree of approximation and repeats the procedure again but with the refined approximation. It is necessary that the precision moves towards the maximum precision to ensure that the search will terminate. The precision is therefore increased each iteration [19]. For a more in dept explanation of UppSAT see Section 2.2.
This report focuses on improving the floating-point to fixed-point approximation presented in the paper *Exploring Approximations for Floating-Point Arithmetic using UppSAT* [19]. The approximation takes formulas using floating-point numbers and translates them into formulas using fixed-point numbers. It uses the theory of fixed length bit-vectors to represent the fixed-point formulas. The approximation uses integers as precision which determines the amount of bits used to represent the fixed-point numbers. The current precision does not enable the possibility to have different amounts of bits for the integral and fractional part of a number. The refinement strategy used in the approximation increases the precision by an arbitrary constant and uses the same precision for the whole formula. This thesis investigates ways to improve this, as well as the precision.

This thesis contributes the following:

- Two new refinement strategies for the approximation.
- A proposal for a new maximum precision.
- An implementation of a simple refinement strategy that leverages a 2D precision.
- An experimental evaluation of the implemented refinement strategy.

## 2 Background

Specifying approximations in UppSAT requires knowledge about SMT and the theories that the approximation uses. Section 2.1 gives a short explanation of SMT and some basic terminology. Section 2.2 explains in more detail the different components of UppSAT and how they interact. The rest of the background explains floating- and fixed-point in Section 2.3 and Section 2.4 respectively.

### 2.1 Satisfiability Modulo Theories

The usual definition of the SAT problem is to find a satisfying assignment for a given formula in propositional logic. A formula in propositional logic uses only Boolean variables and the logical connectives: and ($\land$), or ($\lor$), and negation ($\neg$) [13]. An assignment assigns either true or false to each variable in the formula. A satisfying assignment is an assignment for which the formula evaluates to true. The SAT problem is NP-complete [7].

**Example 1.** The formula $A \land B$ has the satisfying assignment of both $A$ and $B$ being true [12].
Propositional logic can express a lot of properties, but some are easier to express in a richer language [10], and some are not possible to express at all in propositional logic. SMT extends SAT with a background theory or a combination of background theories. A background theory extends the language with a new set of formulas instead of the ones used in SAT [12]. Another way to look at the background theories is that they assign a specific meaning to the symbols used in the formula [2]. An example of a background theory is the theory of integers which allows the use of integers in formulas.

Example 2. $\phi$ is a formula in the theory of integers.

$$\phi = (x < 1) \land (x > 1)$$

With the interpretation of $<$ and $>$ as the usual lesser than and greater than for integers is $\phi$ unsatisfiable, whereas the formula in Example 1 with the same structure is satisfiable.

A solution is sometimes referred to as a model of the formula. The model provides an interpretation of the symbols and variables in the formula that satisfies it [9]. The background theory fixes the interpretation of most symbols in SMT [9]. SMT-solvers are programs that solve instances of the SMT problem [12].

The SMT-LIB standard defines how to specify theories and communicate with SMT-solvers [1]. Two relevant background theories are the theory of fixed length bit-vectors and the theory of floating-point arithmetic. The theory of fixed length bit-vectors enables reasoning about bit-vectors of a fixed length using common operations such as bitwise and, or, and negation as well as basic arithmetic operations interpreting the bit-vectors as binary numbers.\(^1\) The theory of floating-point arithmetic makes it possible to reason about floating-point numbers according to the IEEE floating-point standard [11]. The theory defined in SMT-LIB only allows binary floating-point numbers [4].

\(^1\)http://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml
2.2 UppSAT

An approximation reduces a problem to another one that is close to the original, but hopefully easier to solve. Two common kinds of approximations are over- and under-approximations. The solution to an over-approximation is sometimes not a solution to the original problem. Under-approximations have too small of a solution space. This makes it possible that the solutions to the original problem are not solutions to the approximation. Mixed abstractions are a combination of over- and under-approximations [19]. Figure 1 shows an illustration of the solutions to an over- and under-approximation compared to the original problem.

![Figure 1: An illustration of over- and under-approximations. The left figure shows the solutions to the original problem compared to the solutions to an over-approximation of the problem. The right figure shows the same but for an under-approximation.](image)

It is possible to combine approximations and SMT, to deal with the high complexity of solving instances of the problem expressed in certain background theories. UppSAT accomplishes this by reducing the input formula to a simpler formula that should be easier to solve. It is sometimes possible, based on a model for the approximated formula, to construct a model for the original formula [19]. In regard to the usage of UppSAT it suffices to think of a model as a satisfying assignment.

UppSAT is a framework built in Scala [16] for creating approximating SMT-solvers. It allows approximations that are neither over- or under-approximations [19]. The ability to be able to mix and match different parts of an approximation is an essential part of UppSAT which enables the possibility to rapidly try new kinds of approximations. UppSAT requires the user to implement some specific things to be able to run an approximation. The parts of an approximation that the user has to implement are: the approximation context, the encoding and decoding, reconstruction, and model and proof guided refinement [19].
**Approximation Context**  The approximation context sets the context for the whole approximation. One way of looking at it is that the approximation context defines *what* should happen while the other components define *how* it should happen. To clarify, the approximation context has to define four (4) components, *the input theory, the output theory, the precision domain, and the precision ordering* [19].

- The input theory defines the formulas that it is possible to apply the approximation to.
- The output theory is used to express the approximated formulas (which also can be the same as the input theory).
- The precision domain is the set of elements that indicate how approximated the formula is.
- The precision ordering is an ordering of the elements in the precision domain to help ensure that the approximation will reach the maximum precision.

**Encoding and Decoding**  The encoding specifies how to translate a formula from the input theory to an approximation of the formula in the output theory. To do this the encoding specifies, among other things, what operations from the output theory to use for representing the operations from the input theory. The decoder specifies how to translate a model for the approximated formula back to a model in the input theory [19].

**Reconstruction**  The purpose of the reconstruction is to try to get a model which satisfies the original formula, based on the decoded model for the approximate formula [19].

**Model- and Proof-Guided Refinement**  The model-guided refinement occurs when reconstruction failed. Failure of reconstruction means that it could not make the model of the approximated formula into a model of the original formula. The model-guided refinement will try to increase the precision of the formula to come closer to solving the original formula by approaching full precision semantics. The proof-guided refinement handles the case when the approximate formula is unsatisfiable [19].
UppSAT uses the specified components and an external SMT-solver such as Z3 [8] or MathSAT 4 [6] to form an approximation loop that is the new approximating SMT-solver. One pass through the loop will go as follows: The encoder receives the original formula as input. It encodes the formula and sends it to an external solver.

Depending on the output from the external solver there are two possibilities:
1. The solver could not find a model for the approximated formula, then the next step is proof guided refinement. The proof guided refinement passes a refined formula to the encoder for another iteration of the loop; 2. The solver could find a model for the approximated formula, then that model is passed to the decoder which translates the model back to the sort of the original problem. If the translated model is not a model of the original formula the next step is reconstruction which manipulates the model to try to make it into a model of the original formula. In the case that reconstruction is unsuccessful the reconstructed model is sent to the model-guided refinement that will increase the precision to try to increase the change of getting a model the next iteration. The loop starts over again but with a more precise approximation after the model-guided refinement.

Between the steps which produce a possible model, that model is checked if it satisfy the original formula, to ensure that the solver does not perform any unnecessary calculations. For a visualization of the loop see Figure 2.

---

Figure 2: The different parts of an approximating solver created with UppSAT, and the flow of information between them. The model- and proof-guided refinement are labeled as satRefine and unsatRefine respectively, for readability reasons.
2.3 Floating-Point Arithmetic

The possibility to efficiently represent and manipulate real numbers is important because of the frequent occurrence of them in real-world applications [15]. It is not possible to represent the infinite set of real numbers on a computer with finite memory, so some kind of approximation and rounding is necessary. One of the more popular ways is to use floating-point arithmetic [15].

This thesis will from here on refer to floating-point numbers as floats. A float is of the form $m \cdot \beta^e$ where $\beta$ is the radix, also called the base, $e$ is the exponent, and $m$ is the significand [15]. The decimal point “floats” along $m$ based on the value of the exponent hence the name.

**Example 3.** Two floating-point numbers $D$ and $B$ with radix 10 and 2 respectively.

\[
D = 12345 \cdot 10^{-4} = 1.2345 \\
B = 10001 \cdot 2^{-1} = 1000.1
\]

A specific way to represent floats is as a triple with three elements: sign, biased exponent, and trailing significand $(S, E, T)$. The sign determines the sign of the number, that is if it is positive or negative. $E$ is equal to the exponent with a constant called the bias added to it. The bias makes it possible to represent negative numbers. The trailing significand is of the form $d_0 d_1 \ldots d_n$ and make up the digits of the number [11].

It is possible to encode the triple using $k$ bits. Going from most significant bit to the least significant bit, the first bit is the sign $S$. The next $w$ bits are the biased exponent. The trailing significand takes up the last $t$ bits. Based on the encoded triple, the actual value stored is calculated in one of five possible ways [11]:

- if $E = 2^w - 1$ and $T \neq 0$ then $v = NaN$
- if $E = 2^w - 1$ and $T = 0$ then $v = (-1)^S \times (+\infty)$
- if $1 \leq E \leq 2^w - 2$ then $v = (-1)^S \times 2^{E - \text{bias}} \times (1 + 2^{1-(t+1)} \times T)$.
- if $E = 0$ and $T \neq 0$ then $v = (-1)^S \times 2^\text{emin} \times (0 + 2^1(t+1) \times T)$ i.e a subnormal number the implicit bit is zero.
- $E = 0$ and $T = 0$ then $v = (-1)^S \times (+0)$ signed zero.

The numbers in case three and four are normal and subnormal numbers respectively. The encoding scheme enables the bit to the left of the decimal point in the significand to be implicitly stored and not actually kept in memory. The elements of the triple determines the value of the implicit bit. The normal numbers have an implicit bit of one and the subnormal numbers have an implicit bit of zero [11]. A common implementation is double precision that uses 64 bits to represent a number. The double precision uses eleven bits for $e$, 52 explicitly stored bits for $T$, and one bit for the sign [11].
One of four possible rounding functions determine how the result of an operation is mapped back to a float [14]. Another noteworthy attribute of floats are that they are not uniformly spaced on the number line. The distance between them gets larger and larger the closer to positive and negative infinity you get. Some arithmetic properties that are true for the real numbers are not true for floating-point arithmetic. Addition and multiplication are no longer associative and multiplication does not distribute over addition. The correct rounding modes keep addition and multiplication commutative [15].

2.4 Fixed-Point Arithmetic

Fixed-point numbers in this thesis use a tuple of two integers that represent the amount of bits that are used before and after the decimal point of the number, and a string of digits whose length is equal to the sum of the pair of numbers.

Example 4. The tuple (4,1) and the digit string 10001 represent the binary number 1.0001.

The fixed-point numbers used in the approximation use the binary format and two’s complement to make it possible to have negative fixed-point numbers.

3 Refinement Strategies

This section goes into more detail about how the relevant parts of the approximation work. Section 3.1 discusses how the previous approximation handles the precision and refinement as well as the method of calculating how many bits are needed to represent a float as fixed-point. Section 3.2 explains how the implemented uniform strategy works and Section 3.3 explains the compositional strategy.

The approximation presented in Exploring Approximations for Floating-Point Arithmetic using UppSAT [19] translates the floating-point operations to similar operations on fixed-point numbers. Likewise, each fixed-point operation is translated to one or more operations on bit-vectors.

The normal and subnormal floating-point numbers are turned into fixed-point numbers by first calculating the real number encoded by the float as specified in Section 2.3. The real number is a bit-string with a decimal-point. As discussed in Section 2.4, fixed-point numbers in this thesis are defined by a tuple of two integers and a digit string. The first integer in the tuple is referred to as $i$ for integral, and the second integer is referred to as $f$ for fractional. The corresponding fixed-point number is the $i$ closest bits to the left of the point and the $f$ closest bits to the right of the point. In the case where there are not enough bits in the real number, the fixed-point is padded with zeros to make it into the correct sort.

Positive and negative zero are turned into one zero according to the current sort. Positive and negative infinity are turned into the largest and smallest value respectively. NaN values are not encoded.
Example 5. The approximation translates an addition of two fixed-point numbers that use five bits for both the integral and fractional part, into addition of two bit-vectors of length ten.

The approximation translates numerical values to the closest possible value according to the current precision, which determines the amount of bits to use for the fixed-point numbers. Precision and refinement are the parts of the approximation which this thesis investigates possible approaches for. In other words does the thesis investigate how approximated the formula should be at the start and how much less approximated it should become each iteration.

3.1 Precision and Refinement

Possible modifications of the old implementation the thesis investigates are the maximum precision, the refinement scheme, the precision domain and the kind of precision. The approximating solver falls back to a full precision search when it reaches the maximum precision. For this reason the maximum precision determines for how long the approximation should go on. The refinement scheme defines how much the precision should increase each iteration to move towards the maximum precision. The precision increase can either be a constant or a variable. The precision domain is the set of the valid precisions. The two different kinds of precision are either uniform or compositional. Uniform is the same for the whole formula and compositional is different precision for different parts of the formula.

The model-guided refinement calculates a new precision based on the failed model, the decoded model, and the current precision. The failed model is a model for the approximate formula that is not a model for the original formula and has gone through the decoding stage. The decoded model is a translation of the approximate model to the same sort as the original formula. The failed model is also in the same sort as the original formula. The way to deal with the precision presented in Exploring Approximations for Floating-Point Arithmetic using UppSAT [19] uses non-negative integers as the domain. The precision starts at 8 and has the maximum precision of 50. A precision of $2^p$ indicates that the integral and fractional part of the fixed-point numbers uses $p$ bits each. The current scheme uses a uniform precision and the refinement scheme is to increase the precision by the constant eight each iteration.

The two new proposed ways to increase the precision build on two assumptions about the bits needed to represent a floating-point number as a fixed-point number. UppSAT stores floating-point numbers as the bit-vector triple from Section 2.3. The triple contains the sign, the exponent, and the significand of the number. Based on the values in the triple, the required number of bits is calculated in the following way. The amount of integral bits needed is equal to the unbiased exponent plus one. The amount of fractional bits needed is equal to the position of the least significant one in the significand minus the unbiased exponent.
That this is the amount of bits needed to represent the number without any loss in precision is motivated separately for the integral and the fractional part. For the integral part, the number of bits start at 1 due to the hidden bit. The value of the unbiased exponent is referred to as $E$. In the processes of calculating the value of the float is the significand multiplied with 2 to the power of $E$ which is the same as shifting the number $E$ bits to the left. Therefore, the integral part needs $E$ bits more. To rephrase, the amount of bits needed for the integral part is equal to one for the hidden bit plus $E$ bits for the left shift when calculating the value of the number.

That the amount of fractional bits is correct can be motivated by the fact that any zeros occurring after the least significant one will not change the number, so the fractional part will need the bits of the significand up to and including the least significant one. But because the significand is "shared" between the integral and fractional part and the $E$ first bits are part of the integral part, it is necessary to subtract those bits from the total count.

Example 6. $(S, E, T) = (1, 1000010.00101110...0)$ stores the float $fp = 9.4375$ in IEEE single precision format. The single precision sets the bias to 127. The unbiased exponent is equal to 3. The number of bits needed for the integral part is therefore equal to 4. The position of the least significant one is 7 counting from 1 so the number of fractional bits needed is 4 as well. The fixed-point representation of the number is $1001.0111$ which uses 4 bits for both the fractional and integral part.
3.2 Uniform Refinement Strategy

The failed model that the refinement receives from the reconstruction can have values that could not be represented in the approximate model. The idea behind the uniform refinement is to try to set the precision to the number of bits needed to represent the largest value in the failed model.

The amount of integral and the amount of fractional bits needed to represent a float as fixed point can differ. To deal with this do the uniform refinement change the precision to a pair of two integers. The first number in the pair determines the amount of bits to use for the integral part, the second one does the same for the fractional part. The precision is uniform to enable a fast implementation.

The amount that the precision should be increased with is calculated for each part of the formula that evaluates to a numeric value. Calculate the bits needed as proposed above and set the next precision to the largest value of integral and fractional bits encountered. If the calculated amount of bits is the same as the current precision then both the integral and fractional part is increased by the constant four.

3.3 Compositional Refinement Strategy

The compositional refinement strategy uses a similar strategy as the first one but switches to a compositional precision instead. The switch to compositional precision requires a new way to calculate what precision the different parts of the formula should be set to. This strategy uses the error based refinement defined in An Approximation Framework for Solvers and Decision Procedures [20] to decide what parts of the formula should have their precision increased. The error based refinement works by ranking the terms by how much error they introduce and increase the ones that introduce the most error.

The amount the selected parts of the formula should be increased by is calculated in the same fashion as for the uniform refinement strategy. The precision is also propagated upwards to ensure that no operation has a precision smaller than its operands.
4 Evaluation of Uniform Refinement Strategy

The experimental evaluation tests the performance of uniform refinement by running it with five different combinations of maximum precision on the satisfiable benchmarks in the QF_FP category of benchmarks in SMT-LIB. An AMD Opteron 2220 SE machine, running 64-bit Linux, ran uniform refinement with the different maximum precisions on the benchmarks, it ran the small-floats approximation and the old refinement on the same benchmarks for comparison. The benchmarks had a timeout of five minutes each. The evaluation compares approximations regarding number of total benchmarks solved and the shortest solving time for each benchmark. The following notation is used for readability reasons. FX-x-y means the uniform refinement strategy with maximum precision of x bits for the integral part and y bits for the integral. \((x, y)\) used in the context of a precision means that \(x\) bits is used for the integral part and \(y\) bits is used for the fractional part.

Among the different maximum precisions, 16 bits for the integral part and 40 bits for the fractional part perform the best. It solves 82 of the 163 benchmarks and 14 of them in the shortest time. The second best maximum precision is 32 bits for both the integral and fractional part, with 81 benchmarks solved and the fastest solver in 19 cases. Section 4.1 gives a more detailed comparison of the different precisions. Compared to the old approximation all the tested maximum precisions have a worse performance, solving fewer benchmarks and taking a longer time in general. Section 4.2 gives a more thorough comparison of the different maximum precisions and the old refinement. Section 4.3 compares the performance of the best maximum precision with the small-floats approximation and the old refinement.

In general, uniform refinement did not perform as well as expected. There are likely different reasons behind this. One reason could be that it increases the precision too much each iteration, which makes it reach maximum precision too fast. When the uniform refinement had a maximum precision of \((16, 40)\) it reached it on 38 of the benchmarks, whereas small-floats and the old implementation only reached maximum precision on 2 and 3 benchmarks respectively. Uniform refinement is also sensitive to the model found by the solver. Running the uniform refinement with different versions of z3 as back-end made z3 find a different model for an approximate formula, which caused a change in the solving time by a factor of fifty. The fact that the precision is uniform in combination with a too fast increase may make the approximate formulas to big, which makes it unnecessary to solve them instead of solving the original formula without an approximation.
### 4.1 Comparison of Maximum Precisions

Table 1 shows the results of running uniform refinement with the different maximum precisions. Each column displays the performance of a different maximum precision. Six different attributes are measured to determine the performance: solved benchmarks, benchmarks solved in the shortest time, benchmarks that maximum precision was reached on, average ranking, average amount of iterations needed to solve a benchmark, and how many benchmarks the approximation was the only one able to solve. The first row in Table 1 displays the total number of test cases that was run and the rest of the rows display the six attributes mentioned above in that order.

There are no significant differences in the amount of benchmarks solved. All precisions solve between 75 and 82 benchmarks before timeout. Observing the amount of benchmarks that the uniform refinement reaches maximum precision on show that 16 bits for the fractional part might be too low. The fact that 16 is not suitable for the fractional part can be explained by the fact that FX-16-16 reaches maximum precision on 65 benchmarks whereas FX-16-40 only reaches maximum precision on 38 benchmarks, which indicates that the fractional part had to be increased. FX-64-64 only reach maximum precision on 21 benchmarks and solve 19 benchmarks in the shortest time but solve fewer benchmarks than the best performing ones.

<table>
<thead>
<tr>
<th></th>
<th>FX-16-16</th>
<th>FX-16-40</th>
<th>FX-32-32</th>
<th>FX-40-16</th>
<th>FX-64-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>163</td>
<td>163</td>
<td>163</td>
<td>163</td>
<td>163</td>
</tr>
<tr>
<td>Solved</td>
<td>77</td>
<td>82</td>
<td>81</td>
<td>77</td>
<td>75</td>
</tr>
<tr>
<td>Best Solver</td>
<td>14</td>
<td>14</td>
<td>19</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>Max Precision</td>
<td>65</td>
<td>38</td>
<td>38</td>
<td>66</td>
<td>21</td>
</tr>
<tr>
<td>Avg Rank</td>
<td>4.13</td>
<td>3.84</td>
<td>3.95</td>
<td>4.11</td>
<td>3.93</td>
</tr>
<tr>
<td>Avg Iterations</td>
<td>2.00</td>
<td>2.49</td>
<td>2.40</td>
<td>2.06</td>
<td>2.76</td>
</tr>
<tr>
<td>Only Solver</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: Comparison of running the uniform refinement strategy with different maximum precisions. The two numbers after “FX” in each column indicate how many bits that was used for the integral and fractional bits respectively.

The two best performing maximum precisions are (16, 40) and (32, 32) that solve 82 and 81 test cases respectively. FX-32-32 solves 5 cases faster but does not solve as many benchmarks. Depending on what is most important, the total amount of solved cases or faster solve times on some specific benchmarks can (32, 32) be considered the best maximum precision. Figure 3 shows a scatter plot comparing FX-16-40 and FX-32-32. The crosses are distributed among the diagonal for the most part so there is not a significant difference between (16, 40) and (32, 32) as the maximum precision.
Figure 3: Scatter plot comparing the two best performing maximum precisions. The horizontal axis shows the solving times for the maximum precision (16, 40). The vertical axis shows the maximum precision (32, 32)

### 4.2 Comparison with the Old Implementation

Figure 4 shows a scatter plot comparing the performance of the old refinement and the (16, 40) maximum precision. The old refinement outperforms the uniform refinement on a lot of benchmarks but there are some test cases which the uniform refinement spends a short amount of time on and the old refinement times out. This could be because the uniform refinement stops approximating if it encounters one of the infinities or \( NaN \). Therefore, the previous approximation have to run more iterations before it stops approximating when solving those benchmarks that cannot be solved without special-values.
4.3 Comparison with the Small-Floats Approximation

Table 2 compares the performance of the best performing maximum precision, the old refinement and a different approximation from Exploring Approximations for Floating-Point Arithmetic using UppSAT [19] that stays in the domain of floating-point during approximation but with a smaller amount of bits for each value. The format of Table 2 is the same as Table 1 in Section 4.1. The small-floats approximation solves 99 out of the 163 benchmarks, which is more than both the old and the new which solve 85 and 82 of the benchmarks respectively. The small-floats approximation is also the best performing solver in 52 benchmarks. Figure 5 shows a scatter plot comparing FX-16-40 and the small-floats approximation. The plot shows that the two do not have similar solving times for many benchmarks. There are some test cases that FX-16-40 solves in a short time and small-floats takes longer time to solve.

<table>
<thead>
<tr>
<th></th>
<th>RPFP(z3)</th>
<th>BV(z3)</th>
<th>FX-16-40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>163</td>
<td>163</td>
<td>163</td>
</tr>
<tr>
<td>Solved</td>
<td>99</td>
<td>85</td>
<td>82</td>
</tr>
<tr>
<td>Best Solver</td>
<td>52</td>
<td>25</td>
<td>23</td>
</tr>
<tr>
<td>Max Precision</td>
<td>2</td>
<td>3</td>
<td>38</td>
</tr>
<tr>
<td>Avg Rank</td>
<td>2.27</td>
<td>2.42</td>
<td>2.53</td>
</tr>
<tr>
<td>Avg Iterations</td>
<td>2.02</td>
<td>3.03</td>
<td>2.49</td>
</tr>
<tr>
<td>Only Solver</td>
<td>9</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2: Comparison of the small floats approximation (RPFP(z3)), the old refinement (BV(z3)), and the best performing maximum precision (16, 40)
5 Related Work

A lot of different ways have been proposed for reasoning about floating-point arithmetic.

**Bit-blasting** A common way to solve formulas in the theory of bit-vectors is to use *flattening* to translate the problem into an equivalent one expressed in propositional logic so that SAT-solver can solve it. The process of flattening is also sometimes referred to as bit-blasting. Bit-blasting introduces a lot of new variables to represent each unique bit in the vectors and the possible results [12]. This does not scale so well. An in depth discussion of bit-blasting is given in the book *Decision procedures an algorithmic point of view* [12].

**Abstract conflict driven clause learning** The abstract conflict driven clause learning algorithm (ACDCL) can be used to reason about floating-point numbers. It does not reason about floating-point numbers directly but about intervals instead. ACDCL have shown to be efficient for reasoning about floating-point arithmetic [18]. Martin Brain et al. gives an explanation of how ACDCL solve floating-point formulas in *Deciding floating-point logic with abstract conflict driven clause learning* [3]. UppSAT is capable to use ACDCL as a back-end.

**Other approximations in UppSAT** Aleksandar Zeljic et al. presents other approximations of floating-point arithmetic than the one touched upon here in *Exploring Approximations for Floating-Point Arithmetic using UppSAT* [19]. One approximating using the real numbers and one staying in the theory of floating-point numbers but using a smaller amount of bits.
**Mixed abstractions**  Brillout et al. have created a framework for iteratively approximating a floating-point formula by using mixed approximations [5]. The over approximations evaluate the calculations using all four of the rounding modes, and checks if the usage of the rounding modes satisfy the original formula. The under approximations reduce the bits the significand can use.

**Other methods**  Ramachandran and Wahl propose a strategy for iterative approximation similar to UppSAT and the one by Brillout et al. Their strategy reduces the problem to another theory which is easier to reason about [17].
6 Conclusion & Future Work

This thesis presents two new ways of applying abstractions and approximations to reasoning about floating-point arithmetic. It does so by providing two refinement strategies for approximating floating-point numbers as fixed-point in UppSAT and a new maximum precision for the new strategies. The uniform refinement strategy tries to set the precision for the whole formula to the amount of bits needed to represent the largest number occurring in the failed model. The compositional refinement strategy uses a compositional precision and ranks the terms of the formula by how much error they introduce and then increase the precision of those terms. The ones that introduce the most error have their precision increased by amount calculated the same way as the first strategy. The report also provides an implementation of the first strategy and a performance comparison to the old implementation.

The uniform refinement perform worse than the old refinement strategy but solve some test cases that the old one is not able to solve at all. The new maximum precision is close to the one that is currently being used but reveals it might be necessary to have a larger precision for the fractional part of the precision. Even though the proposed changes does not show a big performance increase do it seems like the precision and refinement can have a big impact on the performance and that using different amount of bits for the integral and fractional part of the precision is the right way to go.

An important task for future work is to actually implement the compositional refinement strategy. It would also be interesting to look into a similar approach to the compositional refinement but increase the precision of the terms by how much error they introduce. There might also be other parts of the current approximation that have room for improvements. For example investigating different ways to deal with the fact that 1023 bits would be needed to represent the largest floating-point number using double precision. Another interesting approach to try with uniform refinement is to set the precision to the median or average of the precision needed for each part of the formula instead of the maximum.
References


