An SMT Theory of Fixed-Point Arithmetic

Marek Baranowski[1], Shaobo He[1(✉)], Mathias Lechner[2], Thanh Son Nguyen[1], and Zvonimir Rakamarić[1]

[1] School of Computing, University of Utah, Salt Lake City, UT, USA
{baranows,shaobo,thanhson,zvonimir}@cs.utah.edu
[2] IST Austria, Klosterneuburg, Austria
mathias.lechner@ist.ac.at

Abstract. Fixed-point arithmetic is a popular alternative to floating-point arithmetic on embedded systems. Existing work on the verification of fixed-point programs relies on custom formalizations of fixed-point arithmetic, which makes it hard to compare the described techniques or reuse the implementations. In this paper, we address this issue by proposing and formalizing an SMT theory of fixed-point arithmetic. We present an intuitive yet comprehensive syntax of the fixed-point theory, and provide formal semantics for it based on rational arithmetic. We also describe two decision procedures for this theory: one based on the theory of bit-vectors and the other on the theory of reals. We implement the two decision procedures, and evaluate our implementations using existing mature SMT solvers on a benchmark suite we created. Finally, we perform a case study of using the theory we propose to verify properties of quantized neural networks.

Keywords: SMT · Fixed-point arithmetic · Decision procedure

1 Introduction

Algorithms based on real arithmetic have become prevalent. For example, the mathematical models in machine learning algorithms operate on real numbers. Similarly, signal processing algorithms often implemented on embedded systems (e.g., fast Fourier transform) are almost always defined over real numbers. However, real arithmetic is not implementable on computer systems due to its unlimited precision. Consequently, we use implementable approximations of real arithmetic, such as floating-point and fixed-point arithmetic, to realize these algorithms in practice.

Floating-point arithmetic is the dominant approximation of real arithmetic that has mature hardware support. Although it enjoys the benefits of being able to represent a large spectrum of real numbers and high precision of arithmetic

This work was supported in part by NSF awards CCF 1552975, CCF 1704715, and the Austrian Science Fund (FWF) under grant Z211-N23 (Wittgenstein Award).

operations over small numbers, floating-point arithmetic, due to its complexity, can be too expensive in terms of speed and power consumption on certain platforms. These platforms are often deployed in embedded systems such as mobile phones, video game consoles, and digital controllers. Recently, the machine learning community revived the interest in fixed-point arithmetic since popular machine learning algorithms and models can be implemented using (even very low bit-width) fixed-points with little accuracy loss [11,27,37]. Therefore, fixed-point arithmetic has been a popular alternative to floating-point arithmetic on such platforms since it can be efficiently realized using integer arithmetic. There are several software implementations of fixed-point arithmetic in different programming languages [22,28,34]; moreover, some programming languages, such as Ada and GNU C, have built-in fixed-point types.

While fixed-point arithmetic is less popular in mainstream applications than floating-point arithmetic, the systems employing the former are often safety-critical. For example, fixed-point arithmetic is often used in medical devices, cars, and robots. Therefore, there is a need for formal methods that can rigorously ensure the correctness of these systems. Although techniques that perform automated verification of fixed-point programs already exist [1,3,15], all of them implement a custom dedicated decision procedure without formalizing the details of fixed-point arithmetic. As a result, it is hard to compare these techniques, or reuse the implemented decision procedures.

On the other hand, ever since the SMT theory of floating-point numbers was formalized [8,44] in SMT-LIB [46], there has been a flurry of research in developing novel and faster decision procedures for the theory [6,7,14,29,35,50]. Meanwhile, the floating-point theory has also been used by a number of approaches that require rigorous reasoning about floating-point arithmetic [2,36,39,41]. The published formalization of the theory enables fair comparison between the decision procedures, sharing of benchmarks, and easy integration of decision procedures within tools that need this functionality. In this paper, we propose and formalize an SMT theory of fixed-point arithmetic, in the spirit of the SMT theory of floating-point arithmetic, with the hope that it will lead to similar outcomes and advances.

Contributions. We summarize our main contributions as follows:

– We present an intuitive and comprehensive syntax of fixed-point arithmetic (Sect. 3) that captures common use cases of fixed-point operations.
– We provide formal semantics of the fixed-point theory based on rational arithmetic (Sect. 4).
– We propose and implement two decision procedures for the fixed-point theory: one that leverages the theory of fixed-size bit-vectors and the other the theory of real numbers (Sect. 5).
– We evaluate the two decision procedures on a set of benchmarks using mature SMT solvers (Sect. 6), and perform a case study of verifying quantized neural networks that uses our theory of fixed-point arithmetic (Sect. 7).

2 Background

Fixed-point arithmetic, like floating-point arithmetic, is used as an approximation for computations over the reals. Both fixed-point and floating-point numbers (excluding the special values) can be represented using rational numbers. However, unlike floating-point numbers, fixed-point numbers in a certain format maintain a fixed divisor, hence the name fixed-point. Consequently, fixed-point numbers have a reduced range of values. However, this format allows for custom precision systems to be implemented efficiently in software—fixed-point arithmetic operations can be implemented in a much smaller amount of integer arithmetic operations compared to their floating-point counterparts. For example, a fixed-point addition operation simply amounts to an integer addition instruction provided that wrap-around is the intended behavior when overflows occur. This feature gives rise to the popularity of fixed-point arithmetic on embedded systems where computing resources are fairly constrained.

A fixed-point number is typically interpreted as a fraction whose numerator is an integer with fixed bit-width in its two's complement representation and denominator is a power of 2. Therefore, a fixed-point format is parameterized by two natural numbers—*tb* that defines the bit-width of the numerator and *fb* that defines the power of the denominator. A fixed-point number in this format can be treated as a bit-vector of length *tb* that is the two's complement representation of the numerator integer and has an implicit binary point between the $fb + 1^{th}$ and fb^{th} least significant bits. We focus on the binary format (as opposed to decimal, etc.) of fixed-point arithmetic since it is widely adopted in hardware and software implementations in practice. Moreover, depending on the intended usage, developers leverage both signed and unsigned fixed-point formats. The signed or unsigned format determines whether the bit pattern representing the fixed-point number should be interpreted as a signed or unsigned integer, respectively. Therefore, signed and unsigned fixed-point formats having the same *tb* and *fb* have different ranges ($[\frac{-2^{tb-1}}{2^{fb}}, \frac{2^{tb-1}-1}{2^{fb}}]$ and $[0, \frac{2^{tb}-1}{2^{fb}}]$), respectively.

Fixed-point addition (resp. subtraction) is typically implemented by adding (resp. subtracting) the two bit-vector operands (i.e., two's complements), amounting to a single operation. Because the denominators are the same between the two operands, we do not need to perform rounding. However, we still have to take care of potential overflows that occur when the result exceeds the allowed range of the chosen fixed-point format. Fixed-point libraries typically implement two methods to handle overflows: saturation and wrap-around. Saturation entails fixing overflowed results to either the minimal or maximal representable value. The advantage of this method is that it ensures that the final fixed-point result is the closest to the actual result not limited by finite precision. Wrap-around allows for the overflowing result to wrap according to two's complement arithmetic. The advantage of this method is that it is efficient and can be used to ensure the sum of a set of (signed) numbers has a correct final value despite potential overflows (if it falls within the supported range). Note that addition is commutative under both methods, but only addition using the wrap-around method is associative. The multiplication and division operations are more involved since they have to include the rounding step as well.

3 Syntax

In this section, we describe the syntax of our proposed theory of fixed-point arithmetic. It is inspired by the syntax of the SMT theory of floating-points [8, 44] and the ISO/IEC TR 18037 standard [23].

Fixed-Points. We introduce the indexed SMT nullary sorts $(\_\ \texttt{SFXP}\ tb\ fb)$ to represent signed fixed-point sorts, where tb is a natural number specifying the total bit-width of the scaled integer in its two's complement form and fb is a natural number specifying the number of fractional bits; tb is greater than or equal to fb. Similarly, we represent unsigned fixed-point sorts with $(\_\ \texttt{UFXP}\ tb\ fb)$. Following the SMT-LIB notation, we define the following two functions for constructing fixed-points literals:

$$((\_\ \texttt{sfxp}\ fb)\ (\_\ \texttt{BitVec}\ tb)\ (\_\ \texttt{SFXP}\ tb\ fb))$$
$$((\_\ \texttt{ufxp}\ fb)\ (\_\ \texttt{BitVec}\ tb)\ (\_\ \texttt{UFXP}\ tb\ fb))$$

where $(\_\ \texttt{sfxp}\ fb)$ (resp. $(\_\ \texttt{ufxp}\ fb)$) produces a function that takes a bit-vector $(\_\ \texttt{BitVec}\ tb)$ and constructs a fixed-point $(\_\ \texttt{SFXP}\ tb\ fb)$ (resp. $(\_\ \texttt{UFXP}\ tb\ fb)$).

Rounding Modes. Similarly to the theory of floating-point arithmetic, we also introduce the `RoundingMode` sort (abbreviated as `RM`) to represent the rounding mode, which controls the direction of rounding when an arithmetic result cannot be precisely represented by the specified fixed-point format. However, unlike the floating-point theory that specifies five different rounding modes, we only adopt two rounding mode constants, namely `roundUp` and `roundDown`, as they are common in practice.

Overflow Modes. We introduce the nullary sort `OverflowMode` (abbreviated as `OM`) to capture the behaviors of fixed-point arithmetic when the result of an operation is beyond the representable range of the used fixed-point format. We adopt two constants, `saturation` and `wrapAround`, to represent the two common behaviors. The `saturation` mode rounds any out-of-bound results to the maximum or minimum values of the representable range, while the `wrapAround` mode wraps the results around similar to bit-vector addition.

Comparisons. The following operators return a Boolean by comparing two fixed-point numbers:

$$\begin{array}{l}
(\texttt{sfxp.geq}\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ \texttt{Bool})\\
(\texttt{ufxp.geq}\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ \texttt{Bool})\\
(\texttt{sfxp.gt}\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ \texttt{Bool})\\
(\texttt{ufxp.gt}\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ \texttt{Bool})\\
(\texttt{sfxp.leq}\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ \texttt{Bool})\\
(\texttt{ufxp.leq}\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ \texttt{Bool})\\
(\texttt{sfxp.lt}\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ \texttt{Bool})\\
(\texttt{ufxp.lt}\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ \texttt{Bool})
\end{array}$$

Arithmetic. We support the following binary arithmetic operators over fixed-point sorts parameterized by tb and fb:

$$\begin{array}{l}
(\texttt{sfxp.add OM}\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ (\texttt{\_ SFXP}\ tb\ fb))\\
(\texttt{ufxp.add OM}\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ (\texttt{\_ UFXP}\ tb\ fb))\\
(\texttt{sfxp.sub OM}\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ (\texttt{\_ SFXP}\ tb\ fb))\\
(\texttt{ufxp.sub OM}\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ (\texttt{\_ UFXP}\ tb\ fb))\\
(\texttt{sfxp.mul OM RM}\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ (\texttt{\_ SFXP}\ tb\ fb))\\
(\texttt{ufxp.mul OM RM}\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ (\texttt{\_ UFXP}\ tb\ fb))\\
(\texttt{sfxp.div OM RM}\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ (\texttt{\_ SFXP}\ tb\ fb)\ \ (\texttt{\_ SFXP}\ tb\ fb))\\
(\texttt{ufxp.div OM RM}\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ (\texttt{\_ UFXP}\ tb\ fb)\ \ (\texttt{\_ UFXP}\ tb\ fb))
\end{array}$$

Note that we force the sorts of operands and return values to be the same. The addition and subtraction operations never introduce error into computation according to our semantics in Sect. 4. Hence, these operators do not take a rounding mode as input like multiplication and division.

Conversions. We introduce two types of conversions between sorts. First, the conversions between different fixed-point sorts:

$$\begin{array}{l}
((\texttt{\_ to\_sfxp}\ tb\ fb)\ \texttt{OM RM}\ (\texttt{\_ SFXP}\ tb'\ fb')\ (\texttt{\_ SFXP}\ tb\ fb))\\
((\texttt{\_ to\_ufxp}\ tb\ fb)\ \texttt{OM RM}\ (\texttt{\_ UFXP}\ tb'\ fb')\ (\texttt{\_ UFXP}\ tb\ fb))
\end{array}$$

Second, the conversions between the real and fixed-point sorts:

$$\begin{array}{l}
((\texttt{\_ to\_sfxp}\ tb\ fb)\ \texttt{OM RM Real}\ (\texttt{\_ SFXP}\ tb\ fb))\\
((\texttt{\_ to\_ufxp}\ tb\ fb)\ \texttt{OM RM Real}\ (\texttt{\_ UFXP}\ tb\ fb))\\
(\texttt{sfxp.to\_real}\ (\texttt{\_ SFXP}\ tb\ fb)\ \texttt{Real})\\
(\texttt{ufxp.to\_real}\ (\texttt{\_ UFXP}\ tb\ fb)\ \texttt{Real})
\end{array}$$

4 Semantics

In this section, we formalize the semantics of the fixed-point theory by treating fixed-points as rational numbers. We first define fixed-points as indexed subsets of rationals. Then, we introduce two functions, rounding and overflow, that are crucial for the formalization of the fixed-point arithmetic operations. Finally, we present the formal semantics of the arithmetic operations based on rational arithmetic and the two aforementioned functions.

Let $\mathbb{F}_{fb} = \{\frac{n}{2^{fb}} \mid n \in \mathbb{Z}\}$ be the infinite set of rationals that can be represented as fixed-points using fb fractional bits. We interpret a signed fixed-point sort (`_ SFXP` tb fb) as the finite subset $\mathbb{S}_{tb,fb} = \{\frac{n}{2^{fb}} \mid -2^{tb-1} \leq n < 2^{tb-1}, n \in \mathbb{Z}\}$ of $\mathbb{F}_{fb}$. We interpret an unsigned fixed-point sort (`_ UFXP` tb fb) as the finite subset $\mathbb{U}_{tb,fb} = \{\frac{n}{2^{fb}} \mid 0 \leq n < 2^{tb}, n \in \mathbb{Z}\}$ of $\mathbb{F}_{fb}$. The rational value of an unsigned fixed-point constant constructed using (`ufxp` bv fb) is $\frac{bv2nat(bv)}{2^{fb}}$, where function $bv2nat$ converts a bit-vector to its unsigned integer value. The rational value of its signed counterpart constructed using (`sfxp` bv fb) is $\frac{bv2int(bv)}{2^{fb}}$, where function $bv2int$ converts a bit-vector to its signed value. Since we treat fixed-point numbers as subsets of rational numbers, we interpret fixed-point comparison operators, such as $=$, `fxp.le`, `fxp.leq`, as simply their corresponding rational comparison relations, such as $=$, $<$, $\leq$, respectively. To be able to formalize the semantics of arithmetic operations, we first introduce the round and overflow helper functions.

We interpret the rounding mode sort `RoundingMode` as the set $rmode = \{ru, rd\}$, where $[\![\texttt{roundUp}]\!] = ru$ and $[\![\texttt{roundDown}]\!] = rd$. Let $rnd_{\mathbb{F}_{fb}} : rmode \times \mathbb{R} \mapsto \mathbb{F}_{fb}$ be a family of round functions parameterized by fb that map a rounding mode and real number to an element of $\mathbb{F}_{fb}$. Then, we define $rnd_{\mathbb{F}_{fb}}$ as

$$\begin{aligned} rnd_{\mathbb{F}_{fb}}(ru, r) &= \min(\{x \mid x \geq r, x \in \mathbb{F}_{fb}\}) \\ rnd_{\mathbb{F}_{fb}}(rd, r) &= \max(\{x \mid x \leq r, x \in \mathbb{F}_{fb}\}) \end{aligned}$$

We interpret the overflow mode sort `OverflowMode` as the set $omode = \{sat, wrap\}$, where $[\![\texttt{saturation}]\!] = sat$ and $[\![\texttt{wrapAround}]\!] = wrap$. Let $ovf_{\mathbb{F}} : omode \times \mathbb{F}_{fb} \mapsto \mathbb{F}$ be a family of overflow functions parameterized by $\mathbb{F}$ that map a rounding mode and element of $\mathbb{F}_{fb}$ to an element of $\mathbb{F}$; here, $\mathbb{F}$ is either $\mathbb{S}_{tb,fb}$ or $\mathbb{U}_{tb,fb}$ depending on whether we are using signed or unsigned fixed-point numbers, respectively. Then, we define $ovf_{\mathbb{F}}$ as

$$ovf_{\mathbb{F}}(sat, x) = \begin{cases} x & \text{if } x \in \mathbb{F} \\ \max(\mathbb{F}) & \text{if } x > \max(\mathbb{F}) \\ \min(\mathbb{F}) & \text{if } x < \min(\mathbb{F}) \end{cases}$$

$$ovf_{\mathbb{F}}(wrap, x) = y \;\text{ such that }\; y \cdot 2^{fb} \equiv x \cdot 2^{fb} \pmod{2^{tb}} \;\wedge\; y \in \mathbb{F}$$

Note that $x \cdot 2^{fb}, y \cdot 2^{fb} \in \mathbb{Z}$ according to the definition of $\mathbb{F}$, and also there is always exactly one y satisfying the constraint.

Now that we introduced our helper round and overflow functions, it is easy to define the interpretation of fixed-point arithmetic operations:

$$\begin{aligned}
[\![\texttt{sfxp.add}]\!](om, x_1, x_2) &= \mathit{ovf}_{\mathbb{S}_{tb,fb}}(om, x_1 + x_2)\\
[\![\texttt{ufxp.add}]\!](om, x_1, x_2) &= \mathit{ovf}_{\mathbb{U}_{tb,fb}}(om, x_1 + x_2)\\
[\![\texttt{sfxp.sub}]\!](om, x_1, x_2) &= \mathit{ovf}_{\mathbb{S}_{tb,fb}}(om, x_1 - x_2)\\
[\![\texttt{ufxp.sub}]\!](om, x_1, x_2) &= \mathit{ovf}_{\mathbb{U}_{tb,fb}}(om, x_1 - x_2)\\
[\![\texttt{sfxp.mul}]\!](om, rm, x_1, x_2) &= \mathit{ovf}_{\mathbb{S}_{tb,fb}}(om, \mathit{rnd}_{\mathbb{F}_{fb}}(rm, x_1 \cdot x_2))\\
[\![\texttt{ufxp.mul}]\!](om, rm, x_1, x_2) &= \mathit{ovf}_{\mathbb{U}_{tb,fb}}(om, \mathit{rnd}_{\mathbb{F}_{fb}}(rm, x_1 \cdot x_2))\\
[\![\texttt{sfxp.div}]\!](om, rm, x_1, x_2) &= \mathit{ovf}_{\mathbb{S}_{tb,fb}}(om, \mathit{rnd}_{\mathbb{F}_{fb}}(rm, x_1 / x_2))\\
[\![\texttt{ufxp.div}]\!](om, rm, x_1, x_2) &= \mathit{ovf}_{\mathbb{U}_{tb,fb}}(om, \mathit{rnd}_{\mathbb{F}_{fb}}(rm, x_1 / x_2))
\end{aligned}$$

Note that it trivially holds that $\forall x_1, x_2 \in \mathbb{F}_{fb} \,.\, x_1 + x_2 \in \mathbb{F}_{fb} \wedge x_1 - x_2 \in \mathbb{F}_{fb}$. Therefore, we do not need to round the results of the addition and subtraction operations. In the case of division by zero, we adopt the semantics of other SMT theories such as reals: $(=\ x\ (\texttt{sfxp.div}\ om\ rm\ y\ 0))$ and $(=\ x\ (\texttt{ufxp.div}\ om\ rm\ y\ 0))$ are satisfiable for every $x, y \in \mathbb{F}$, $om \in \mathit{omode}$, $rm \in \mathit{rmode}$. Furthermore, for every $x, y \in \mathbb{F}$, $om \in \mathit{omode}$, $rm \in \mathit{rmode}$, if $(=\ x\ y)$ then $(=\ (\texttt{sfxp.div}\ om\ rm\ x\ 0)\ (\texttt{sfxp.div}\ om\ rm\ y\ 0))$ and $(=\ (\texttt{ufxp.div}\ om\ rm\ x\ 0)\ (\texttt{ufxp.div}\ om\ rm\ y\ 0))$.

Note that the order of applying the *rnd* and *ovf* functions to the results in real arithmetic matters. We choose *rnd* followed by *ovf* since it matches the typical real-world fixed-point semantics. Conversely, reversing the order can lead to out-of-bound results. For example, assume that we extend the signature of the *ovf* function to $\mathit{omode} \times \mathbb{R} \mapsto \mathbb{R}$ while preserving its semantics as a modulo operation over 2^{tb-fb}. Then, $\mathit{ovf}_{\mathbb{U}_{3,2}}(\mathit{wrap}, 7.5)$ evaluates to $\frac{7.5}{4}$, and applying $\mathit{rnd}_{\mathbb{F}_2}$ to it when the rounding mode is ru evaluates to $\frac{8}{4}$; this is greater than the maximum number in $\mathbb{U}_{3,2}$, namely $\frac{7}{4}$. On the other hand, evaluating $\mathit{ovf}_{\mathbb{U}_{3,2}}(\mathit{wrap}, \mathit{rnd}_{\mathbb{F}_2}(\mathit{ru}, 7.5))$ produces 0, which is the expected result. We could apply the *ovf* function again to the out-of-bound results, but the current semantics achieves the same without this additional operation.

Let $\mathit{cast}_{\mathbb{F},\mathbb{F}_{fb}} : \mathit{omode} \times \mathit{rmode} \times \mathbb{R} \mapsto \mathbb{F}$ be a family of cast functions parameterized by $\mathbb{F}$ and $\mathbb{F}_{fb}$ that map an overflow mode, rounding mode, and real number to an element of $\mathbb{F}$; as before, $\mathbb{F}$ is either $\mathbb{S}_{tb,fb}$ or $\mathbb{U}_{tb,fb}$ depending on whether we are using signed or unsigned fixed-point numbers, respectively. Then, we define $\mathit{cast}_{\mathbb{F},\mathbb{F}_{fb}}(om, rm, r) = \mathit{ovf}_{\mathbb{F}}(om, \mathit{rnd}_{\mathbb{F}_{fb}}(rm, r))$, and the interpretation of the conversions between reals and fixed-points as

$$\begin{aligned}
[\![(\texttt{\_ to\_sfxp}\ tb\ fb)]\!](om, rm, r) &= \mathit{cast}_{\mathbb{S}_{tb,fb},\mathbb{F}_{fb}}(om, rm, r)\\
[\![(\texttt{\_ to\_ufxp}\ tb\ fb)]\!](om, rm, r) &= \mathit{cast}_{\mathbb{U}_{tb,fb},\mathbb{F}_{fb}}(om, rm, r)\\
[\![\texttt{sfxp.to\_real}]\!](r) &= r\\
[\![\texttt{ufxp.to\_real}]\!](r) &= r
\end{aligned}$$

5 Decision Procedures

In this section, we propose two decision procedures for the fixed-point theory by leveraging the theory of fixed-size bit-vectors in one and the theory of reals in the other.

Bit-Vector Encoding. The decision procedure based on the theory of fixed-size bit-vectors is akin to the existing software implementations of fixed-point arithmetic that use machine integers. More specifically, a fixed-point sort parameterized by tb is encoded as a bit-vector sort of length tb. Therefore, the encoding of the constructors of fixed-point values simply amounts to identity functions. Similarly, the encoding of the comparison operators uses the corresponding bit-vector relations. For example, the comparison operator `sfxp.lt` is encoded as `bvslt`. The essence of the encoding of the arithmetic operations is expressing the numerator of the result, after rounding and overflow handling, using bit-vector arithmetic. We leverage the following two observations in our encoding. First, rounding a real value v to the value in the set $\mathbb{F}_{fb}$ amounts to rounding $v \cdot 2^{fb}$ to an integer following the same rounding mode. This observation explains why rounding is not necessary for the linear arithmetic operations. Second, we can encode the wrap-around of the rounded result as simply extracting tb bits from the encoded result thanks to the wrap-around nature of the two's complement SMT representation. We model the behavior of division-by-zero using uninterpreted functions of the form (`RoundingMode` `OverflowMode` (`_ BitVec` tb) (`_ BitVec` tb)), with one such function for each fixed-point sort appearing in the query. The result of division-by-zero is then the result of applying this function to the numerator, conditioned on the denominator being equal to zero. This ensures that all divisions-by-zero with equal numerators produce equal results when the overflow and rounding modes are also equal.

Real Encoding. The decision procedure based on the theory of reals closely mimics the semantics defined in Sect. 4. We encode all fixed-point sorts as the real sort, while we represent fixed-point values as rational numbers. Therefore, we can simply encode fixed-point comparisons as real relations. For example, both `sfxp.lt` and `ufxp.lt` are translated into $<$ relation. We rely on the first observation above to implement the rounding function rnd_{fb} using an SMT real-to-integer conversion. We implement the overflow function $ovf_{tb,fb}$ using the SMT remainder function. Note that the encodings of both functions involve non-linear real functions, such as the real-to-int conversion. Finally, we model division as the rounded, overflow-corrected result of the real theory's division operation. Since the real theory's semantics ensures that equivalent division operations produce equivalent results, this suffices to capture the fixed-point division-by-zero semantics.

Implementation. We implemented the two decision procedures within the pySMT framework [25]: the two encodings are rewriting classes of pySMT. We made our implementations publicly available.[1] We also implemented a random generator of queries in our fixed-point theory, and used it to perform thorough differential testing of our decision procedures.

6 Experiments

We generated the benchmarks we use to evaluate the two encodings described in Sect. 5 by translating the SMT-COMP non-incremental QF_FP benchmarks [45]. The translation accepts benchmarks that contain only basic arithmetic operations defined in both theories. Moreover, we exclude all the benchmarks in the *wintersteiger* folder because they are mostly simple regressions to test the correctness of an implementation of the floating-point theory. In the end, we manage to translate 218 QF_FP benchmarks in total.

We translate each QF_FP benchmark into 4 benchmarks in the fixed-point theory, which differ in the configurations of rounding and overflow modes. We denote a configuration as a (rounding mode, overflow mode) tuple. Note that changing a benchmark configuration alters the semantics of its arithmetic operations, which might affect its satisfiability. Our translation replaces floating-point sorts with fixed-point sorts that have the same total bit-widths; the number of fractional bits is half of the bit-width. This establishes a mapping from single-precision floats to Q16.16 fixed-points implemented by popular software libraries such as libfixmath [34]. It translates arithmetic operations into their corresponding fixed-point counterparts using the chosen configuration uniformly across a benchmark. The translation also replaces floating-point comparison operations with their fixed-point counterparts. Finally, we convert floating-point constants by treating them as reals and performing real-to-fixed-point casts. We made our fixed-point benchmarks publicly available.[2]

The SMT solvers that we use in the evaluation are Boolector [9] (version 3.1.0), CVC4 [4] (version 1.7), MathSAT [13] (version 5.5.1), Yices2 [19] (version 2.6.1), and Z3 [17] (version 4.8.4) for the decision procedure based on the theory of bit-vectors. For the evaluation of the decision procedure based on the theory of reals, we use CVC4, MathSAT, Yices2, and Z3. We ran the experiments on a machine with four Intel E7-4830 sockets, for a total of 32 physical cores, and 512GB of RAM, running Ubuntu 18.04. Each benchmark was limited to 1200 s of wall time and 8GB of memory, and no run of any benchmark exceeded the memory limit. We set processor affinity for each solver instance in order to reduce variability due to cache effects.

Table 1 shows the results of running the SMT solvers on each configuration with both encodings (bit-vector and real). We do not observe any inconsistencies in terms of satisfiability reported among all the solvers and between both encodings. The performance of the solvers on the bit-vector encoding is typically better than on the real encoding since it leads to fewer timeouts and crashes. Moreover, all the solvers demonstrate similar performance for the bit-vector encoding

[1] https://github.com/soarlab/pysmt/tree/fixed-points.
[2] https://github.com/soarlab/QF_FXP.

Table 1. The results of running SMT solvers on the four different configurations of the benchmarks using both encodings. Boolector and MathSAT are denoted by Btor and MSAT, respectively. Column "All" indicates the number of benchmarks for which any solver answered sat or unsat; benchmarks for which no solver gave an answer are counted as unknown.

(a) (RoundUp, Saturation)

Result	Bit-Vector Encoding					Real Encoding				All
	Btor	CVC4	MSAT	Yices2	Z3	CVC4	MSAT	Yices2	Z3	
sat	57	52	47	65	43	15	50	52	22	65
unsat	129	127	127	125	131	125	126	126	124	132
timeout	32	39	44	28	44	75	15	40	37	
unknown	0	0	0	0	0	3	0	0	35	21
error	0	0	0	0	0	0	27	0	0	

(b) (RoundUp, WrapAround)

Result	Bit-Vector Encoding					Real Encoding				All
	Btor	CVC4	MSAT	Yices2	Z3	CVC4	MSAT	Yices2	Z3	
sat	58	51	53	67	48	14	34	52	14	72
unsat	128	128	126	128	134	123	65	124	121	134
timeout	32	39	39	23	36	79	102	42	60	
unknown	0	0	0	0	0	2	0	0	23	12
error	0	0	0	0	0	0	17	0	0	

(c) (RoundDown, Saturation)

Result	Bit-Vector Encoding					Real Encoding				All
	Btor	CVC4	MSAT	Yices2	Z3	CVC4	MSAT	Yices2	Z3	
sat	59	52	54	62	50	29	53	57	22	64
unsat	128	127	127	125	134	127	130	130	124	135
timeout	31	39	37	31	34	58	11	31	46	
unknown	0	0	0	0	0	4	0	0	26	19
error	0	0	0	0	0	0	24	0	0	

(d) (RoundDown, WrapAround)

Result	Bit-Vector Encoding					Real Encoding				All
	Btor	CVC4	MSAT	Yices2	Z3	CVC4	MSAT	Yices2	Z3	
sat	57	65	54	67	50	23	39	55	14	71
unsat	128	128	127	129	133	125	81	90	121	134
timeout	33	25	37	22	35	68	65	73	57	
unknown	0	0	0	0	0	2	0	0	26	13
error	0	0	0	0	0	0	33	0	0	

Table 2. Comparison of the number of benchmarks (considering all configurations) solved by a solver but not solved by another solver. Each row shows the number of benchmarks solved by the row's solver but not solved by the column's solver. We mark the bit-vector (resp. real) encoding with B (resp. R).

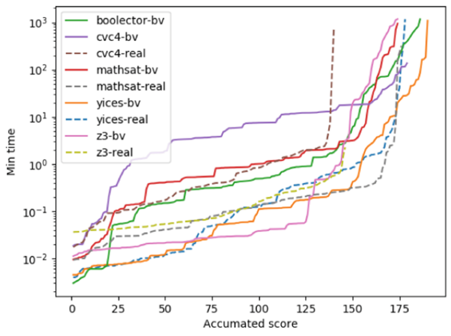
	Btor-B	CVC4-B	MSAT-B	Yices2-B	Z3-B	CVC4-R	MSAT-R	Yices2-R	Z3-R
Btor-B	—	33	37	11	52	165	183	86	185
CVC4-B	19	—	46	6	57	154	160	70	170
MSAT-B	8	31	—	4	39	141	174	78	160
Yices2-B	35	44	57	—	79	194	198	95	208
Z3-B	31	50	47	34	—	151	189	103	168
CVC4-R	2	5	7	7	9	—	113	49	41
MSAT-R	17	8	37	8	44	110	—	23	118
Yices2-R	28	26	49	13	66	154	131	—	162
Z3-R	3	2	7	2	7	22	102	38	—

across all the configurations, whereas they generally produce more timeouts for the real encoding when the overflow mode is wrap-around. We believe that this can be attributed to the usage of nonlinear operations (e.g., real to int casts) in the handling of wrap-around behaviors. This hypothesis could also explain the observation that the bit-vector encoding generally outperform the real encoding when the overflow mode is wrap-around since wrap-around comes at almost no cost for the bit-vector encoding (see Sect. 5).

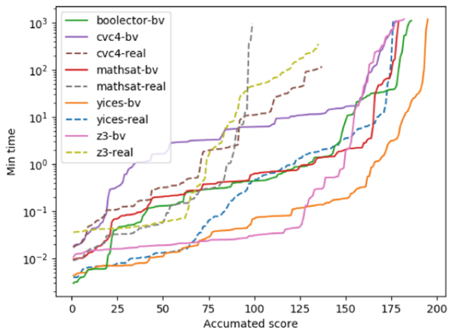
Column "All" captures the performance of the solvers when treated as one portfolio solver. This improves the overall performance since the number of solved benchmarks increases, indicating that each solver has different strengths and weaknesses. Table 2 further analyzes this behavior, and we identify two reasons for it when we consider unique instances solved by each individual solver. First, when the overflow mode is saturation, Yices2 is the only solver to solve unique instances for both encodings. Second, when the overflow mode is wrap-around, the uniquely solved instances come from solvers used on the bit-vector encoding, except one that comes from Yices2 on the real encoding. These results provide further evidence that the saturation configurations are somewhat easier to solve with reals, and that wrap-around is easier with bit-vectors.

Figure 1 uses quantile plots [5] to visualize our experimental results in terms of runtimes. A quantile plot shows the minimum runtime on y-axis within which each of the x-axis benchmarks is solved. Some characteristics of a quantile plot are helpful in analyzing the runtimes. First, the rightmost x coordinate is the number of benchmarks that a solver returns meaningful results for (i.e., sat or unsat). Second, the uppermost y coordinate is the maximum runtime of all the benchmarks. Third, the area under a line approximates the total runtime.

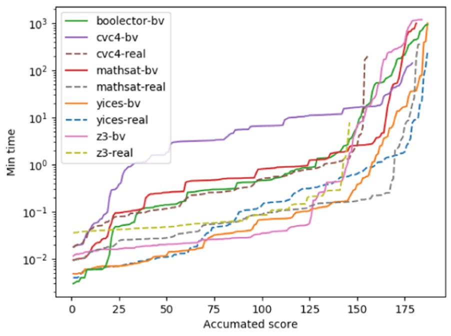
Although the semantics of the benchmarks vary for each configuration, we can observe that the shapes of the bit-vector encoding curves are similar, while those of the real encoding differ based on the chosen overflow mode. More precisely, solvers tend to solve benchmarks faster when their overflow mode is saturation



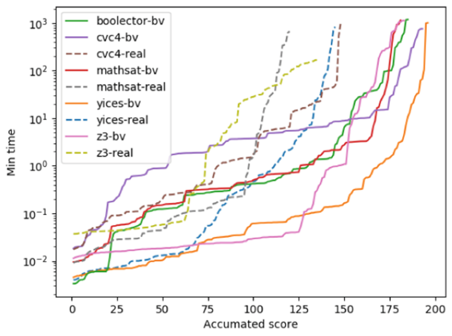
(a) (RoundUp, Saturation)



(b) (RoundUp, WrapAround)



(c) (RoundDown, Saturation)



(d) (RoundDown, WrapAround)

Fig. 1. Quantile plots of our experimental results.

as opposed to wrap-around. We observe the same behavior in Table 1, and it is likely due to the fact that we introduce nonlinear operations to handle wrap-around behaviors when using the real encoding.

7 Case Study: Verification of Quantized Neural Networks

Neural networks have experienced a significant increase in popularity in the past decade. Such networks that are realized by a composition of non-linear layers are able to efficiently solve a large variety of previously unsolved learning tasks. However, neural networks are often viewed as black-boxes, whose causal structure cannot be interpreted easily by humans [40]. This property makes them unfit for applications where guaranteed correctness has a high priority. Advances in formal methods, in particular SMT solvers, leveraging the piece-wise linear structure of neural networks [20,31,47], have made it possible to verify certain formal properties of neural networks of reasonable size. While these successes provide an essential step towards applying neural networks to safety-critical tasks, these

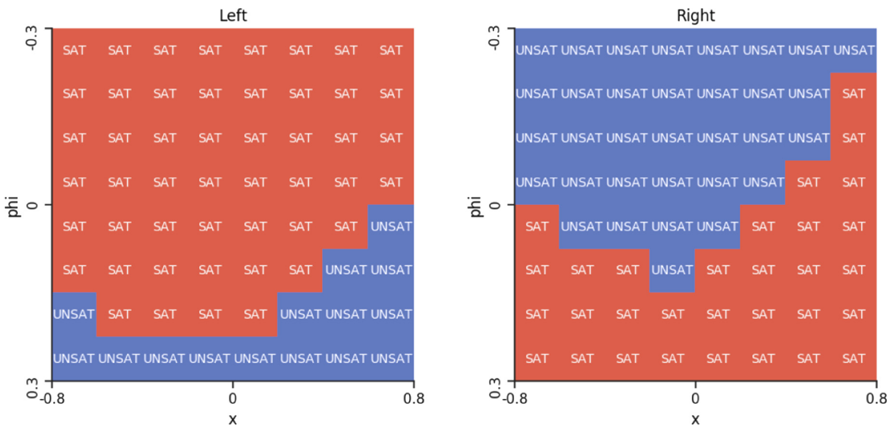


Fig. 2. Satisfiability of specifications of our cart-pole controller.

methods leave out one crucial aspect—neural networks are usually *quantized* before being deployed to production systems [30].

Quantization converts a network that operates over 32-bit floating-point semantics into a fewer-bit fixed-point representation. This process serves two goals: compressing the memory requirement and increasing the computational efficiency of running the network. Quantization introduces additional non-linear rounding operations to the semantics of a neural network. Recently, Giacobbe et al. [26] have shown that, in practice, this can lead to situations where a network that satisfies formal specifications might violate them after the quantization step. Therefore, when checking formal properties of quantized neural networks, we need to take their fixed-point semantics into account.

We derive a set of example fixed-point problem instances based on two machine learning tasks to demonstrate the capabilities of our fixed-point SMT theory on realistic problems. For all tasks, we train multi-layer perceptron modules [43] with ReLU-7 activation function [32] using quantization-aware training [30]. This way we avoid that quantization results in a considerable loss of accuracy. To encode a neural network into an SMT formula, we rely on the Giacobbe et al.'s [26] approach for encoding the summations and activation functions. We quantize all neural networks using the signed fixed-point format with 8 bits total and 4 fractional bits. We are using the bit-vector encoding decision procedure in combination with the Boolector SMT solver.

7.1 Cart-Pole Controller

In our first task, we train a neural network controller using the cart-pole environment of OpenAI's "gym" reinforcement learning suite. In this task, an agent has to balance a pole mounted on a movable cart in an upright position. The cart provides four observation variables x,$\dot{x}$,φ,$\dot{\varphi}$ to the controller, where x is the position of the cart and φ the angle of the pole. The controller then steers the cart by discrete actions (move left or right). Our neural network agent, composed of three layers (4,8,1), solves the task by achieving an average score of the maximal 500 points. We analyze what our black-box agent has learned by using our decision procedure. In particular, we are interested in how much our agent relies on the input variable x compared to φ for making a decision. Moreover, we are interested in which parts of the input space the agent's decision is constant. We assume the dynamics of the cart is bounded, i.e., $-0.3 \leq \dot{x} \leq 0.3, -0.02 \leq \dot{\varphi} \leq 0.2$, and partition the input space of the remaining two input variables into a grid of 64 tiles. We then check for each tile whether there exists a situation when the network would output a certain action (left, right) by invoking our decision procedure.

Figure 2 shows that the agent primarily relies on the input variable φ for making a decision. If the angle of the pole exceeds a certain threshold, the network is guaranteed to make the vehicle move left; on the other hand, if the angle of the pole is below a different threshold, the network moves the vehicle right. Interestingly, this pattern is non-symmetric, despite the task being entirely symmetric.

7.2 Checking Fairness

For our second task, we checked the fairness specification proposed by Giacobbe et al. [26] to evaluate the maximum influence of a single input variable on the decision of a network. We train a neural network on student data to predict the score on a math exam. Among other personal features, the gender of a person is fed into the network for making a decision. As the training data contains a bias in the form of a higher average math score for male participants, the network might learn to underestimate the math score of female students. We employ our decision procedure to compute the maximum influence of the gender of a person to its predicted math score. First, we create encodings of the same network (3 layers of size 6, 16, and 1) that share all input variables except the gender as a single fixed-point theory formula. We then constrain the predicted scores such that the one network outputs a score that is c higher than the score predicted by the other network. Finally, we perform binary search by iteratively invoking our decision procedure to find out at what bias c the formula changes from satisfiable to unsatisfiable.

Table 3. Satisfiability of specifications of our fairness example.

Score Diff	Status	Runtime
11.25	sat	10s
11.5	sat	9s
11.75	unsat	200s
12	unsat	706s

Table 3 shows that there exists a hypothetical person whose predicted math score would drop by 11.5 points out of 100 if the person is female instead of male. Moreover, our results also show that for no person the math score would change by 11.75 points if the gender would be changed.

8 Related Work

Ruemmer and Wahl [44] and Brain et al. [8] propose and formalize the SMT theory of the IEEE-754 floating-point arithmetic. We were inspired by these papers both in terms of the syntax and the formalization of the semantics of our theory. There are several decision procedures for the floating-point theory. In particular, Brain et al. [7] present an efficient and verified reduction from the theory of floating-points to the theory of bit-vectors, while Leeser et al. [33] solve the floating-point theory by reducing it to the theory of reals. These two decision procedures are much more complicated than the ones we describe in Sect. 5 due to the more complex nature of floating-point arithmetic.

In the rest of this section, we introduce related approaches that perform verification or synthesis of programs that use fixed-point arithmetic. Many of these approaches, and in particular the SMT-based ones, could benefit from our unified formalization of the theory of fixed-point arithmetic. For example, they could leverage our decision procedures instead of developing their own from scratch. Moreover, having the same format allows for easier sharing of benchmarks and comparison of results among different decision procedures.

Eldib et al. [21] present an SMT-based method for synthesizing optimized fixed-point computations that satisfy certain acceptance criteria, which they rigorously verify using an SMT solver. Similarly to our paper, their approach encodes fixed-point arithmetic operations using the theory of bit-vectors. Anta et al. [3] tackle the verification problem of the stability of fixed-point controller implementations. They provide a formalization of fixed-point arithmetic semantics using bit-vectors, but unlike our paper they do not formalize rounding and overflows. Furthermore, they encode the fixed-point arithmetic using unbounded integer arithmetic, arguing that unbounded integer arithmetic is a better fit for their symbolic analysis. We could also reduce our bit-vector encoding to unbounded integers following a similar scheme as Anta et al.

Bounded model checker ESMBC [15,24] supports fixed-point arithmetic and has been used to verify safety properties of fixed-point digital controllers [1]. Like us, it also employs a bit-vector encoding. However, it is unclear exactly which fixed-point operations are supported. UppSAT [50] is an approximating SMT solver that leverages fixed-point arithmetic as an approximation theory to floating-point arithmetic. Like the aforementioned work, UppSAT also encodes fixed-point arithmetic using the theory of bit-vectors. Its encoding ignores rounding modes, but adds special values such as infinities.

In addition to SMT-based verification, another important aspect of reasoning about fixed-point computations is error bound analysis, which is often used for the synthesis of fixed-point implementations. Majumdar et al. [38] synthesize Pareto optimal fixed-point implementations of control software in regard to performance criteria and error bounds. They reduce error bound computation to an optimization problem solved by mixed-integer linear programming. Darulova et al. [16] compile real-valued expressions to fixed-point expressions, and rigorously show that the generated expressions satisfy given error bounds. The error bound analysis is static and based on affine arithmetic. Volkova et al. [48,49] propose an approach to determine the fixed-point format that ensures the absence of overflows and minimizes errors; their error analysis is based on Worst-Case Peak Gain measure. TAFFO [12] is an LLVM plugin that performs precision tuning by replacing floating-point computations with their fixed-point counterparts. The quality of precision tuning is determined by a static error propagation analysis.

9 Conclusions and Future Work

In this paper, we propose an SMT theory of fixed-point arithmetic to facilitate SMT-based software verification of fixed-point programs and systems by promoting the development of decision procedures for the proposed theory. We introduce the syntax of fixed-point sorts and operations in the SMT-LIB format similar to that of the SMT floating-point theory. Then, we formalize the semantics of the fixed-point theory, including rounding and overflow, based on the exact rational arithmetic. We develop two decision procedures for the fixed-point theory that encode it into the theory of bit-vectors and reals. Finally, we study the performance of our prototype decision procedures on a set of benchmarks, and perform a realistic case study by proving properties of quantized neural networks.

As future work, we plan to add more complex operations to the fixed-point theory, such as conversions to/from floating-points and the remainder operation. Moreover, we would like to apply the fixed-point theory to verify existing software implementations of fixed-point arithmetic in different programming languages. We plan to do this by integrating it into the Boogie intermediate verification language [18] and the SMACK verification toolchain [10,42].

References

1. Abreu, R.B., Gadelha, M.Y.R., Cordeiro, L.C., de Lima Filho, E.B., da Silva, W.S.: Bounded model checking for fixed-point digital filters. J. Braz. Comput. Soc. **22**(1), 1:1–1:20 (2016). https://doi.org/10.1186/s13173-016-0041-8
2. Andrysco, M., Nötzli, A., Brown, F., Jhala, R., Stefan, D.: Towards verified, constant-time floating point operations. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS), pp. 1369–1382 (2018). https://doi.org/10.1145/3243734.3243766
3. Anta, A., Majumdar, R., Saha, I., Tabuada, P.: Automatic verification of control system implementations. In: Proceedings of the International Conference on Embedded Software (EMSOFT), pp. 9–18 (2010). https://doi.org/10.1145/1879021.1879024
4. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
5. Beyer, D.: Software verification and verifiable witnesses. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 401–416. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_31

6. Brain, M., D'Silva, V., Griggio, A., Haller, L., Kroening, D.: Deciding floating-point logic with abstract conflict driven clause learning. Formal Methods Syst. Des. **45**(2), 213–245 (2013). https://doi.org/10.1007/s10703-013-0203-7
7. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 79–98. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_5
8. Brain, M., Tinelli, C., Rümmer, P., Wahl, T.: An automatable formal semantics for IEEE-754 floating-point arithmetic. In: Proceedings of the IEEE International Symposium on Computer Arithmetic (ARITH), pp. 160–167 (2015). https://doi.org/10.1109/ARITH.2015.26
9. Brummayer, R., Biere, A.: Boolector: an efficient SMT solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_16
10. Carter, M., He, S., Whitaker, J., Rakamarić, Z., Emmi, M.: SMACK software verification toolchain. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 589–592 (2016). https://doi.org/10.1145/2889160.2889163
11. Cherkaev, A., Tai, W., Phillips, J.M., Srikumar, V.: Learning in practice: reasoning about quantization. CoRR abs/1905.11478 (2019). http://arxiv.org/abs/1905.11478
12. Cherubin, S., Cattaneo, D., Chiari, M., Bello, A.D., Agosta, G.: TAFFO: tuning assistant for floating to fixed point optimization. Embed. Syst. Lett. **12**(1), 5–8 (2020). https://doi.org/10.1109/LES.2019.2913774
13. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
14. Conchon, S., Iguernlala, M., Ji, K., Melquiond, G., Fumex, C.: A three-tier strategy for reasoning about floating-point numbers in SMT. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 419–435. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_22
15. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 137–148 (2009). https://doi.org/10.1109/ASE.2009.63
16. Darulova, E., Kuncak, V., Majumdar, R., Saha, I.: Synthesis of fixed-point programs. In: Proceedings of the International Conference on Embedded Software (EMSOFT), pp. 22:1–22:10 (2013). https://doi.org/10.1109/EMSOFT.2013.6658600
17. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
18. DeLine, R., Leino, K.R.M.: BoogiePL: a typed procedural language for checking object-oriented programs. Technical report, MSR-TR-2005-70, Microsoft Research (2005)
19. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
20. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 269–286. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_19

21. Eldib, H., Wang, C.: An SMT based method for optimizing arithmetic computations in embedded software code. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **33**(11), 1611–1622 (2014). https://doi.org/10.1109/TCAD.2014.2341931
22. The fixed crate. https://gitlab.com/tspiteri/fixed
23. Programming languages – C – extensions to support embedded processors. Standard 18037, ISO/IEC (2008). https://www.iso.org/standard/51126.html
24. Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: an industrial-strength C model checker. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 888–891. https://doi.org/10.1145/3238147.3240481
25. Gario, M., Micheli, A.: PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In: International Workshop on Satisfiability Modulo Theories (SMT) (2015)
26. Giacobbe, M., Henzinger, T.A., Lechner, M.: How many bits does it take to quantize your neural network? In: Biere, A., Parker, D. (eds.) TACAS 2020. LNCS, vol. 12079, pp. 79–97. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_5
27. Gupta, S., Agrawal, A., Gopalakrishnan, K., Narayanan, P.: Deep learning with limited numerical precision. In: Proceedings of the International Conference on Machine Learning (ICML), pp. 1737–1746 (2015)
28. Signed 15.16 precision fixed-point arithmetic. https://github.com/ekmett/fixed
29. He, S., Baranowski, M., Rakamarić, Z.: Stochastic local search for solving floating-point constraints. In: Zamani, M., Zufferey, D. (eds.) NSV 2019. LNCS, vol. 11652, pp. 76–84. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28423-7_5
30. Jacob, B., et al.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2704–2713 (2018). https://doi.org/10.1109/CVPR.2018.00286
31. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
32. Krizhevsky, A.: Convolutional deep belief networks on CIFAR-10 (2010, unpublished manuscript)
33. Leeser, M., Mukherjee, S., Ramachandran, J., Wahl, T.: Make it real: effective floating-point reasoning via exact arithmetic. In: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 1–4 (2014). https://doi.org/10.7873/DATE.2014.130
34. Cross platform fixed point maths library. https://github.com/PetteriAimonen/libfixmath
35. Liew, D., Cadar, C., Donaldson, A.F., Stinnett, J.R.: Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In: Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 521–532 (2019). https://doi.org/10.1145/3338906.3338921
36. Liew, D., Schemmel, D., Cadar, C., Donaldson, A.F., Zähl, R., Wehrle, K.: Floating-point symbolic execution: a case study in n-version programming. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 601–612 (2017). https://doi.org/10.1109/ASE.2017.8115670

37. Lin, D.D., Talathi, S.S., Annapureddy, V.S.: Fixed point quantization of deep convolutional networks. In: Proceedings of the International Conference on Machine Learning (ICML), pp. 2849–2858 (2016)
38. Majumdar, R., Saha, I., Zamani, M.: Synthesis of minimal-error control software. In: Proceedings of the International Conference on Embedded Software (EMSOFT), pp. 123–132 (2012). https://doi.org/10.1145/2380356.2380380
39. Menendez, D., Nagarakatte, S., Gupta, A.: Alive-FP: automated verification of floating point based peephole optimizations in LLVM. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 317–337. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_16
40. Olah, C., et al.: The building blocks of interpretability. Distill (2018). https://doi.org/10.23915/distill.00010
41. Paganelli, G., Ahrendt, W.: Verifying (in-)stability in floating-point programs by increasing precision, using SMT solving. In: Proceedings of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 209–216 (2013). https://doi.org/10.1109/SYNASC.2013.35
42. Rakamarić, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 106–113. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_7
43. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**(6088), 533–536 (1986). https://doi.org/10.1038/323533a0
44. Rümmer, P., Wahl, T.: An SMT-LIB theory of binary floating-point arithmetic. In: Informal Proceedings of the International Workshop on Satisfiability Modulo Theories (SMT) (2010)
45. SMT-LIB benchmarks in the QF_FP theory. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_FP
46. SMT-LIB: the satisfiability modulo theories library. http://smtlib.cs.uiowa.edu
47. Tjeng, V., Xiao, K.Y., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: International Conference on Learning Representations (ICLR) (2019)
48. Volkova, A., Hilaire, T., Lauter, C.: Determining fixed-point formats for a digital filter implementation using the worst-case peak gain measure. In: Proceedings of the Asilomar Conference on Signals, Systems and Computers, pp. 737–741 (2015). https://doi.org/10.1109/ACSSC.2015.7421231
49. Volkova, A., Hilaire, T., Lauter, C.Q.: Arithmetic approaches for rigorous design of reliable fixed-point LTI filters. IEEE Trans. Comput. **69**(4), 489–504 (2020). https://doi.org/10.1109/TC.2019.2950658
50. Zeljić, A., Backeman, P., Wintersteiger, C.M., Rümmer, P.: Exploring approximations for floating-point arithmetic using UppSAT. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 246–262. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_17