Oracle Integration of Floating-Point Solvers with Isabelle

Olle Torstensson and Tjark Weber

August 11, 2022

Oracle Integration of Floating-Point Solvers with Isabelle

Olle Torstensson[1][2] and Tjark Weber[2]

[1] Linköping University, Linköping, Sweden
`olle.torstensson@liu.se`
[2] Uppsala University, Uppsala, Sweden
`tjark.weber@it.uu.se`

Abstract

Sledgehammer, a component of the interactive proof assistant Isabelle, aims to increase proof automation by automatically discharging proof goals with the help of external provers. Among these provers are a group of satisfiability modulo theories (SMT) solvers with support for the SMT-LIB input language. Despite existing formalizations of IEEE floating-point arithmetic in both Isabelle/HOL and SMT-LIB, Sledgehammer employs an abstract translation of floating-point types and constants, depriving the SMT solvers of the opportunity to make use of their dedicated decision procedures for floating-point arithmetic.

We show that, by extending the translation from the language of Isabelle/HOL into SMT-LIB with an interpretation of floating-point types and constants, floating-point reasoning in SMT solvers can be made available to Isabelle. Our main contribution is a description and implementation of such an extension. An evaluation of the extended translation shows a significant increase of Sledgehammer's success rate on proof goals involving floating-points. In particular, this enhancement enables Sledgehammer to prove more non-trivial goals—thereby increasing proof automation for floating-point arithmetic in Isabelle.

1 Introduction

Interactive theorem proving is one of the more flexible and powerful formal verification techniques available. However, finding a proof outline with intermediate proof steps just simple enough for a proof assistant to be able to discharge automatically may require a considerable amount of time and effort, even from a seasoned user. As an example, the seL4 micro-kernel, the product of about two person-years and 9000 lines of code, took a total of about 20 person-years and 200,000 lines of proof development to formally verify [27]. For this reason, increasing proof automation in interactive proof assistants is crucial to further broaden their applicability.

As a way of tackling this issue, many interactive proof assistants have the ability of transferring the proof burden of some of the intermediate steps onto *automated* reasoning systems with automatic proof methods better suited for the task. This approach has proven to be quite successful in bringing the number of required user interactions down for many types of problems, thus increasing productivity.

Among these proof assistants, we find Isabelle [30] and its powerful proof-delegation tool Sledgehammer [32], which acts as an interface between Isabelle/HOL and a number of external provers. In addition to "traditional" (resolution-based) first-order automatic theorem provers (ATPs) such as E [36], SPASS [42], and Vampire [34], these external provers include satisfiability modulo theories (SMT) solvers like Z3 [16], CVC4 [6], and veriT [12]. SMT solvers are highly specialized in reasoning within certain logical theories (e.g., integers, real numbers, and bit vectors), and often implement decision procedures more efficient than those found in the automatic proof methods of Isabelle. The dedicated inference rules of the SMT solvers, however, are accessible to Isabelle only when the types and constants that appear in the delegated proof obligations are interpreted in the language of the SMT solver. An abstract translation that leaves types and constants uninterpreted will deprive external solvers of the opportunity to make use of their dedicated decision procedures for specific background theories, and will instead have to rely on the right set of facts being passed to the solver along with the conjecture.

One of the more recent additions to the growing set of theories supported by most major SMT solvers via the satisfiability modulo theories library (SMT-LIB) standard [5], is that of floating-point arithmetic [13]. A formalization of IEEE floating-point arithmetic in Isabelle/HOL has been available in the Archive of Formal Proofs for nearly a decade [43]. Sledgehammer has, however, not yet caught up to this development; its SMT component does not implement an interpretation of floating-point types and constants. Our aim is to provide such an interpretation, with the purpose of increasing the success rate for floating-point proof obligations delegated to SMT solvers, and thereby to increase the degree of automation in the interactive proof process.

Contributions. We extend the SMT solver integration in Isabelle by adding support for floating-point arithmetic, i.e., by treating floating-point types and operations as interpreted in the translation from the language of Isabelle/HOL to the SMT-LIB input format. In addition to describing this extension in detail (Section 3), we provide an implementation[1] aimed toward Sledgehammer. To the best of our knowledge, this enhancement makes Isabelle the first interactive proof assistant to get this kind of support. An evaluation (Section 4), performed on a small set of floating-point conjectures, confirms the expectation that our translation extension significantly increases Sledgehammer's success rate on proof goals involving floating-point arithmetic.

This new part of the SMT solver integration in Isabelle functions as an oracle; proofs found by an SMT solver are not reconstructed internally, but need to be trusted. Proof reconstruction remains as recommended work for the future (see Section 6).

2 Background

In this section, we cover the necessary theory concerning Sledgehammer and floating-point arithmetic.

2.1 The Sledgehammer Proof Process

When trying to prove a conjecture in Isabelle, a user may, via a simple call to Sledgehammer, pass along the proof obligation to several external provers, which will then work on the problem in parallel. The statement to be proven is used by a relevance filter [28] to find additional facts (axioms and already proven statements) that may help the with the proving process. All of these statements are then translated and compiled into a file in the input format of the external prover (in the case of SMT solvers, an SMT-LIB input file), as illustrated in Figure 1.

After working on the problem, the prover returns to Isabelle with its findings. At this point, if a prover reported the conjecture to be true, the user can either choose to view the prover as an *oracle* and accept the conjecture as a theorem (the dashed path in Figure 1), or make Isabelle try to automatically reconstruct the proof internally, based on the additional facts sent with the conjecture and any proof details the prover may provide. Theorems that are only proved

[1]Our implementation is currently available at `https://github.com/olletorstensson/floathammer`. We plan to submit a more final version to the Archive of Formal Proofs.

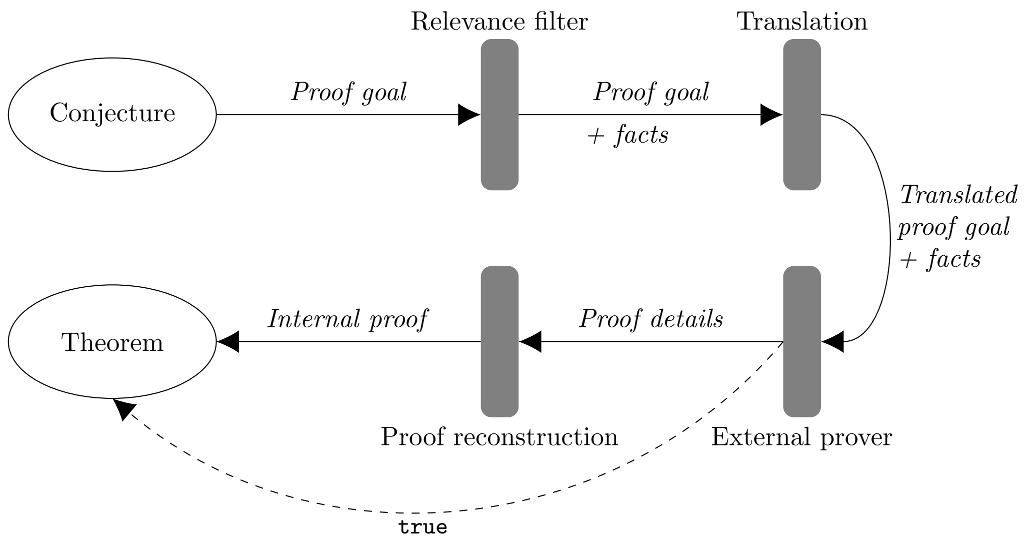

Figure 1: A conjecture's journey to become a theorem via Sledgehammer.

externally are marked with an *oracle* tag, meant to convey a certain amount of skepticism—reconstructed proofs are generally preferred, as they remove the consideration of possible bugs in the external prover, or in the translation between formats.

In Sledgehammer's translation module, types and constants are generally declared with a unique (freshly generated) identifier and (for constants) a sort that have no inherent meaning to the external prover. A few Isabelle theories (e.g., those for integer arithmetic, real arithmetic, and bit vectors) define types and constants that are treated as interpreted by the translation into SMT-LIB [9], in which case they are mapped directly to their counterpart in the target logic—thereby allowing the SMT solvers to use their built-in decision procedures designed specifically to reason within the theories in question.

2.2 IEEE 754 Binary Floating-Point Arithmetic

The most common way to approximate the real numbers to a suitable finite set of numbers in modern hardware is via *floating-points*. Simulating real arithmetic using floating-points is not a straightforward task; the definitions of arithmetic operations are not always obvious, and should ideally not vary between implementations. To this end, the IEEE developed their technical standard IEEE 754 [24], aiming to provide clear specifications and recommendations on all aspects of floating-point arithmetic. To meet the needs of different applications, the standard specifies several floating-point *formats*, each defining a unique set of numbers.

A binary floating-point format is characterized by its exponent width $w \in \mathbb{N}$, and its precision $p \in \mathbb{N}$. A binary floating-point number, x, may then be represented in this format by a triple (s, e, f) of bit vectors of length 1, w, and $p-1$, respectively, such that (for finite x)

$$x = \begin{cases} (-1)^s \cdot 2^{1-\mathrm{bias}(w)} \cdot (0 + \frac{f}{2^{p-1}}) & \text{if } \; e = 0 \\ (-1)^s \cdot 2^{e-\mathrm{bias}(w)} \cdot (1 + \frac{f}{2^{p-1}}) & \text{otherwise,} \end{cases}$$

where $\text{bias}(w) = 2^{w-1} - 1$. Together, the sign s, the (biased) exponent e, and the fraction f constitute a unique representation of any finite or infinite floating-point number; in particular, the two numbers $+0$, represented by $(0, 0 \dots 0, 0 \dots 0)$, and -0, represented by $(1, 0 \dots 0, 0 \dots 0)$, are considered distinct. The standard also specifies two signed infinities—denoted by $+\infty$ and $-\infty$, and represented by $(0, 1 \dots 1, 0 \dots 0)$ and $(1, 1 \dots 1, 0 \dots 0)$, respectively—representing values that are too great in magnitude for the format, together with *Not-a-Number* (NaN), representing undefined values via any triple $(s, 1 \dots 1, f)$ such that $f \neq 0 \dots 0$.[2]

Additionally, IEEE 754 specifies various arithmetic operations on floating-point numbers. Conceptually, floating-point arithmetic is carried out by converting floating-point numbers to more precise values, performing the corresponding arithmetic operation, and converting the result back to the original floating-point format, in an emulation of a rounded but infinitely precise calculation. In a setting such as Isabelle/HOL, where theories of real arithmetic are available, the task of carrying out calculations with infinite precision falls upon these, whereas the floating-point operations handle the rounding and special cases (e.g., when an argument is NaN or infinite). IEEE 754 specifies precisely how this handling should be performed.

3 A Translation of Floating-Point Arithmetic from Isabelle/HOL to SMT-LIB

This section describes an interpreted translation of floating-point types and operations from Isabelle/HOL to SMT-LIB. Our translation extends a preexisting *general* translation [9] targeting SMT solvers that is part of Sledgehammer, which treats floating-point arithmetic as uninterpreted. It supports the formal model of IEEE floating-point arithmetic in Isabelle/HOL (originally by Lei Yu) that is available in the Archive of Formal Proofs [43]. We aim to be comprehensive but restrict attention to those floating-point concepts that are defined in both theories.

3.1 SMT-LIB Logic

The first task of our translation module is to select an SMT-LIB logic within which the SMT solver is to reason when deciding the satisfiability of the formula. For performance reasons, it is generally a good idea to select a logic that is as specific as allowed by the contents and structure of the formula. However, `FP`, the logic dedicated to floating-points, is too restrictive for many of Isabelle's proof obligations, which may contain types and constants that, when translated, require support for symbols that are either free (uninterpreted) or declared in a different theory than that of floating-points.

Sledgehammer's SMT integration relies on callback functions to analyze the proof obligation and determine the problem's logic. However, only one of these functions may select a logic. In the absence of a framework allowing for a more modular approach (e.g., incrementally generalizing the logic as little as necessary, whenever it is required), translations utilizing the interpretation of floating-point concepts will have to be paired with the most general logic available. To achieve this, whenever a floating-point specific type is detected as a subtype in a term of the formula to be translated, our callback function selects the logic `ALL`, which instructs the SMT solver to use the most general logic that it supports.

[2]The IEEE 754 standard defines two "types" of NaN, a *quiet* and a *signalling* NaN. Neither the Isabelle theory nor the SMT-LIB theory make this distinction as they are formalizations based on a higher level of abstraction.

3.2 Types

Both theories implement binary floating-point formats of arbitrary width of the exponent and fraction fields. In Isabelle, `(m, n) float` is the type of floating-points with an exponent field of width `m`, and a fraction field of width `n` (and thus with precision `n+1`). In SMT-LIB, the hidden bit of the significand (the bit preceding the fraction) is included in the format specification, making `(_ FloatingPoint m n+1)` the corresponding sort.[3] Due to the lack of support for higher-order features in SMT-LIB, there is no sort corresponding to Isabelle's polymorphic floating-point type, and the interpretation has to be limited to floating-point types with arguments encoding fixed numeric values (see recommendations for future work in Section 6).

In addition to the types/sorts implementing the floating-point data structures, the two theories each contain a small enumerated type/sort for the rounding modes used by the arithmetic operations. To make those isomorphic, the Isabelle rounding mode type has been supplemented with an additional rounding mode (not imposed by IEEE 754, but defined in SMT-LIB) for *round to nearest, ties away from zero.*

Non-isomorphism. The floating-point types in Isabelle are defined, via isomorphisms, as sets of triples of bit vectors of certain lengths. For any $k \in \mathbb{N}$, let $\mathbf{BV}_k$ be the set of bit vectors of length k. Then the type `(m, n) float` is isomorphic to the set

$$\mathbf{FP}^{\mathbf{ISA}}_{\mathtt{m,n}} \coloneqq \mathbf{BV}_1 \times \mathbf{BV}_{\mathtt{m}} \times \mathbf{BV}_{\mathtt{n}}.$$

The definition of the floating-point sorts in SMT-LIB is not quite as straightforward. For each m and n, the theory defines four disjoint sets:

$$\begin{aligned}
\mathbf{Z}_{m,n} &\coloneqq \left\{ (s,e,f) \in \mathbf{BV}_1 \times \mathbf{BV}_m \times \mathbf{BV}_n \mid e = \mathbf{0}_m \wedge f = \mathbf{0}_n \right\} \\
\mathbf{S}_{m,n} &\coloneqq \left\{ (s,e,f) \in \mathbf{BV}_1 \times \mathbf{BV}_m \times \mathbf{BV}_n \mid e = \mathbf{0}_m \wedge f \neq \mathbf{0}_n \right\} \\
\mathbf{N}_{m,n} &\coloneqq \left\{ (s,e,f) \in \mathbf{BV}_1 \times \mathbf{BV}_m \times \mathbf{BV}_n \mid e \neq \mathbf{0}_m \wedge e \neq \mathbf{1}_m \right\} \\
\mathbf{I}_{m,n} &\coloneqq \left\{ (s,e,f) \in \mathbf{BV}_1 \times \mathbf{BV}_m \times \mathbf{BV}_n \mid e = \mathbf{1}_m \wedge f = \mathbf{0}_n \right\},
\end{aligned}$$

where, for any $k \in \mathbb{N}$, $\mathbf{0}_k$ and $\mathbf{1}_k$ are the bit vectors consisting of k zeros and k ones, respectively. The union of these sets, together with a special NaN value, is the logical interpretation of the SMT-LIB floating-point sort `(_ FloatingPoint m n+1)`:

$$\mathbf{FP}^{\mathbf{SMT}}_{\mathtt{m,n+1}} \coloneqq \mathbf{Z}_{\mathtt{m,n}} \cup \mathbf{S}_{\mathtt{m,n}} \cup \mathbf{N}_{\mathtt{m,n}} \cup \mathbf{I}_{\mathtt{m,n}} \cup \{\text{NaN}\}.$$

In the above listing of sets making up $\mathbf{FP}^{\mathbf{SMT}}_{m,n+1}$, one set of bit vector triples is notably absent. In the Isabelle theory, all elements of the set

$$\mathbf{FP}^{\mathbf{ISA}}_{m,n} \setminus \mathbf{FP}^{\mathbf{SMT}}_{m,n+1} = \left\{ (s,e,f) \in \mathbf{BV}_1 \times \mathbf{BV}_m \times \mathbf{BV}_n \mid e = \mathbf{1}_m \wedge f \neq \mathbf{0}_n \right\}$$

are considered distinct as floating-points, and they all denote NaN, whereas the SMT-LIB theory instead introduces a single special NaN value with no bit-vector representation. As a result, $\mathbf{FP}^{\mathbf{ISA}}_{m,n}$ and $\mathbf{FP}^{\mathbf{SMT}}_{m,n+1}$ are not isomorphic whenever $m > 1$ and $n > 1$. Although both theories take some liberties with the IEEE 754 standard due to theorem provers not being its main target for implementations, it seems that neither theory violates the standard in this regard. The

[3]The SMT-LIB sorts are only defined for formats with $\mathtt{m} > 1$ and $\mathtt{n} > 0$, whereas `m` and `n` are merely required to be positive in Isabelle. Thus, any type `(1, n) float` lacks a corresponding sort in SMT-LIB, and has to be left uninterpreted by the translation.

SMT-LIB theory explicitly states that it is based on *floating-point data* ("specification level 2" in IEEE 754), while the Isabelle theory is based on bit-vector *representations* of floating-point data. This issue and its impact on the soundness of the translation is further discussed in Section 3.4—and a possible remedy, in Section 6.

3.3 Constants

For the sake of brevity, we focus here on some of the more interesting aspects of the translation of constants. For the interested reader, a more detailed description of the translation is available elsewhere [39]. An exhaustive enumeration of the mapping is provided in Table 1.

Polymorphism. The issue regarding polymorphism, described in the previous section, affects the translation of constants as well. A constant can only be interpreted if its type is not polymorphic. Since Isabelle's automatic type inference assigns constants the most general type possible with respect to the context, variables and constants with a floating-point type will in general need to be attached with explicit type constraints in order to trigger the interpretation.

Direct correspondence. For many floating-point related constants in Isabelle, there is a direct semantic-preserving mapping to a function in SMT-LIB. Among these we find, e.g., the rounding modes and comparison operations together with many arithmetic operations and classification predicates. The translation of these does not involve much more than simply replacing the identifier with the corresponding one in SMT-LIB. The classification predicates for "NaN-ity" (`is_nan` in Isabelle, and `fp.isNaN` in SMT-LIB) define non-isomorphic sets (due to the underlying sets of floating-points being non-isomorphic), but do correspond in their definition; they both decide whether a floating-point does not belong to any other floating-point category:

$$\begin{aligned}\left\{ x \in \mathbf{FP}^{\mathbf{ISA}}_{m,n} \mid \texttt{is\_nan}(x) \right\} &= \mathbf{FP}^{\mathbf{ISA}}_{m,n} \setminus (\mathbf{Z}_{m,n} \cup \mathbf{S}_{m,n} \cup \mathbf{N}_{m,n} \cup \mathbf{I}_{m,n}) \\ \left\{ x \in \mathbf{FP}^{\mathbf{SMT}}_{m,n+1} \mid \texttt{fp.isNaN}(x) \right\} &= \mathbf{FP}^{\mathbf{SMT}}_{m,n+1} \setminus (\mathbf{Z}_{m,n} \cup \mathbf{S}_{m,n} \cup \mathbf{N}_{m,n} \cup \mathbf{I}_{m,n}).\end{aligned}$$

Therefore, even this mapping can be applied directly.

Format parameter extraction. A few SMT-LIB functions targeted by our translation are technically elements of an infinite set of functions generated by an index over all floating-point formats. This holds, e.g., for the conversion operation from reals to floating-points, and for the (nullary) functions denoting the special floating-point values $+0$, $+\infty$ and NaN, as their sort is not necessarily derivable from context. In these cases, an extra step, in which we extract the type arguments of the type of the constant to be translated and add them explicitly as arguments to the corresponding function symbol in SMT-LIB, is required for a correct interpretation. For instance, the Isabelle constant `some_nan`, acting as the canonical NaN of type `float m n`, is mapped to the single NaN `(_ NaN m n+1)` of sort `(_ FloatingPoint m n+1)` whenever `m` and `n` encode fixed numeric values.

Term translation. Some subtle differences in how certain constants in Isabelle and functions in SMT-LIB are defined force us to go beyond a mere symbol-to-symbol mapping. As a first example, `fp`—the function in SMT-LIB defining floating-point values from bit vectors—is the corresponding *curried* version of Isabelle's `Abs_float`. In order to correctly translate formulas involving `Abs_float`, the function and its argument cannot be considered separately, but rather

Table 1: Types and constants in Isabelle, together with sorts and functions in SMT-LIB covered by the translation. `m` > 1 and `n` > 0 denote the format of the type/sort. The variables s, e, and f denote bit vectors of length 1, ≥ 1, and ≥ 1, respectively. Double entries in the Isabelle column indicate that term rewriting is necessary for the latter. Square brackets denote syntactic sugar, which is also interpreted.

	Isabelle	Smt-Lib
Floating-Point Type	`(m, n) float`	`(_ FloatingPoint m n+1)`
Rounding Mode Type	`roundmode`	`RoundingMode`
Value Construction	`Abs_float` (s, e, f)	`(fp` s e f`)`
Positive Infinity	`plus_infinity` [∞]	`(_ +oo m n+1)`
Not-a-Number	`some_nan`	`(_ NaN m n+1)`
Positive Zero	`zero_class.zero [0]`	`(_ +zero m n+1)`
Round to Nearest[1]	`To_nearest`	`RNE`
Round to Nearest[2]	`To_nearest_away`[3]	`RNA`
Round Upward	`To_pinfinity`	`RTP`
Round Downward	`To_ninfinity`	`RTN`
Round Toward Zero	`float_To_zero`	`RTZ`
Zeroness	`is_zero`	`fp.isZero`
Infinity	`is_infinity`	`fp.isInfinite`
NaN-ity	`is_nan`	`fp.isNaN`
Normality	`is_normal`	`fp.isNormal`
Denormality	`is_denormal`	`fp.isSubnormal`
Absolute Value	`abs` [$\lvert\_\rvert$]	`fp.abs`
Negation	`uminus [-]`	`fp.neg`
Addition	`fadd` / `plus [+]`	`fp.add`
Subtraction	`fsub` / `minus [-]`	`fp.sub`
Multiplication	`fmul` / `times [*]`	`fp.mul`
Division	`fdiv` / `divide [div]`	`fp.div`
Fused Mult. and Add.	`fmul_add`	`fp.fma`
Remainder	`float_rem` / `frem`[4]	`fp.rem`
Square Root	`fsqrt` / `float_sqrt`	`fp.sqrt`
Integral Rounding	`fintrnd` / `ROUNDFLOAT`	`fp.roundToIntegral`
Exclusive Less-Than	`flt [<]`	`fp.lt`
Inclusive Less-Than	`fle` [$\leq$]	`fp.leq`
Exclusive Greater-Than	`fgt`	`fp.gt`
Inclusive Greater-Than	`fge`	`fp.geq`
Equality	`feq` [$\doteq$]	`fp.eq`
From Real	`round`[5]	`(_ to_fp m n+1)`
To Real	`valof`	`fp.to_real`

[1] Ties to even.
[2] Ties away from zero.
[3] New addition.
[4] Modified (error fixed).
[5] Modified to accommodate new rounding mode.

they have to be translated directly as a composite term. The constant `Abs_float` is then interpreted as `fp`, while its argument—a triple of bit vectors in Isabelle/HOL—is split into three separate bit-vector arguments, taken care of by the preexisting interpretation of bit vectors.

A number of arithmetic operations in the Isabelle theory have an alternate version that adheres to IEEE's recommendation of letting *round to nearest, ties to even* be "the default rounding-direction attribute for results in binary formats" [24] by setting the rounding mode to `To_nearest`. All functions on the SMT-LIB side that implement these operations do however require a rounding mode to be explicitly passed as an argument—except for the remainder function. As noted in both the IEEE 754 standard and in the SMT-LIB theory, the remainder function is exact, and thereby it does not require rounding to produce a floating-point result. To be able to interpret the versions of the arithmetic operations not directly corresponding with their SMT-LIB counterpart, the translation first performs some term rewriting, where a rounding mode is added (or in the case of the remainder function, removed) as necessary.

Modifications to the Isabelle theory. After supplementing the rounding-mode datatype in Isabelle with an additional element, some additional changes to the theory were required to accommodate this new rounding mode. Isabelle's implementation of floating-point arithmetic relies heavily on the conversion operations to real numbers, so these modifications were limited to the two operations `round` and `intround` (the latter of which is not interpreted in our translation), and merely amounted to adding another case for the new rounding mode in each operation, while carefully considering the overflow thresholds and return values specified by IEEE 754.

An error in the remainder function `frem` as defined in the Isabelle theory was discovered during implementation and has been patched: the remainder of a finite x and $\pm\infty$ shall be x [24].[4]

3.4 (Un-)Soundness

While the definition of the rounding-mode type has been altered in order to unify the two theories, the definition of the floating-point type in Isabelle/HOL is currently left unchanged. As long as this discrepancy between the two theories remains, our translation is unsound, i.e., statements that are false in Isabelle may translate into statements that are true in SMT-LIB. Consider, for instance, the following formula in Isabelle (where all floating-points are assumed to have a *fixed* floating-point type, which we suppress for sake of readability):

$$\texttt{is\_nan x} \wedge \texttt{is\_nan y} \implies \texttt{x = y}.$$

This is not a valid theorem in Isabelle/HOL, as witnessed by, e.g., `x` = `some_nan` and `y` = `-x`, but when passing the problem along to Sledgehammer, the supported SMT solvers report the translated statement to be true—because, according to the SMT-LIB floating-point theory (which has only one NaN element), it is. Note that `=` is the regular equality of Isabelle/HOL, comparing arguments as equal only if they have the same bit-vector representation, as opposed to IEEE 754 equality of the floating-point numbers they represent.

We note that the problem pertains in its essence to the different semantics of NaN between the two theories. The translation is therefore sound for certain (important) classes of formulas: ground formulas that do not contain a term denoting NaN, and formulas that only contain NaN-*uniform* operations, i.e., operations that return the same result for all NaNs regardless of

[4]The modified theory is currently available at https://github.com/olletorstensson/floathammer.

their bit-vector representation (just like all functions in SMT-LIB do). Many of the floating-point operations defined in Isabelle are in fact NaN-uniform (but `sign`, `fraction`, `abs`, `uminus`, `valof`, and `=` are notable exceptions).

Another source of unsoundness is the direct mapping of `valof`, Isabelle's conversion operation from floating-points to real numbers, to SMT-LIB's `fp.to_real`. Whereas they correspond on finite floating-points, the result of applying `fp.to_real` to $\pm\infty$ or NaN is unspecified and may vary between different SMT solvers.[5]

Evidently, the consequence of the unsoundness introduced by the interpretation of floating-point types and constants is an oracle that cannot be trusted. This is indeed unfortunate, but it is only a temporary state; once the Isabelle theory has undergone the necessary modifications, trust should be restored, and once proof reconstruction has been implemented, trust is no longer an issue (see Section 6).

4 Evaluation

To investigate the difference in the performance of Sledgehammer brought on by the translation, and to get a clear overview of the comparative performance of the SMT solvers, we conducted an experimental evaluation on a small set of floating-point related proof-obligations. Freely available Isabelle formalizations of floating-point properties are scarce; only a few properties are included in the theories accompanying the main Isabelle floating-point theory. We complemented these with our own formalizations of floating-point properties taken from the IEEE 754 standard and the *Handbook of Floating-point Arithmetic* [29], resulting in a set of 123 formulas.[6] The formulas in the evaluation set exhibit difficulties ranging from trivial to levels on par with Sterbenz' lemma [38], and they are all expressed in such a way that they have a sound translation.

The generated proof obligations were passed along one by one to external tools via Sledgehammer with default settings (including a 30 second time limit). The default external tools invoked by Sledgehammer in Isabelle2021 are the ATPs E (version 2.5.1), SPASS (version 3.8), and Vampire (version 4.2.2), along with the SMT solvers CVC4 (version 1.8), veriT (version 2020.10), and Z3 (version 4.4.0). Out of the three SMT solvers, only Z3 offers unrestricted support for floating-points; veriT does not support the floating-point theory at all, and CVC4 only allows floating-point sorts denoting either the single or double precision format specified by IEEE 754. Whereas the restrictions of CVC4 can be circumvented by enabling its "experimental mode" (presumably at the cost of less reliable results), veriT is rendered useless when faced with a problem involving interpreted floating-point types or constants.

The experiments were conducted in Isabelle2021 under macOS version 10.15, running on a dual-core 2.5 GHz processor machine with 16 GB of RAM.

4.1 Results

The evaluation set was run for a few fixed floating-point formats, meant to give an estimate of the performance for arbitrary (fixed) formats. The fixed formats chosen for this task were the half, single, double, and quadruple precision formats specified by IEEE 754. As to not

[5]One solution to this problem could be to map `valof` to a new SMT-LIB function defined to produce the same values as `fp.to_real` on finite inputs and the same values as `valof` on all other inputs. This approach requires more substantial modifications to the way SMT-LIB inputs are generated within Isabelle/HOL and is currently not part of our implementation.

[6]The full evaluation set is currently available at https://github.com/olletorstensson/floathammer.

disregard the possibility of a better performance with more general results by means of polymorphism, the evaluation set was run also for the polymorphic format with interpretation disabled. This gives rise to nine different *models* (technically, Isabelle theories with different type annotations) for success measurement of the provers on the evaluation set, defined for $\texttt{x} \in \{\texttt{(5,10)}, \texttt{(8,23)}, \texttt{(11,52)}, \texttt{(15,112)}\}$ as:

- $\mathbb{U}_\texttt{x}$: Interpretation is disabled and all floating-points are of type `x float`.
- $\mathbb{I}_\texttt{x}$: Interpretation is enabled and all floating-points are of type `x float`.
- $\mathbb{U}_\text{poly}$: Interpretation is disabled and all floating-points are of type `('e, 'f) float`.

For convenience, the four fixed formats are abbreviated by their total bit length (16, 32, 64, and 128, respectively) in the model names.

In addition to the models above, six aggregate models, used to estimate the success for arbitrary interpreted floating-point types, are defined for $X \in \{\mathbb{I}, \mathbb{U}\}$ as:

- X_some: A goal is successfully proven if it is so in *at least one* of the models X_{16}, X_{32}, X_{64}, and X_{128}.
- X_all: A goal is successfully proven if it is so in *all* of the models X_{16}, X_{32}, X_{64}, and X_{128}.
- X_mean: The average success in the models X_{16}, X_{32}, X_{64}, and X_{128}.

Table 2 shows the success rates for four different prover configurations of Sledgehammer when run directly on the evaluation set in the models described above. In addition to the default solver configuration ALL, an abbreviation for CVC4 + E + SPASS + Vampire + veriT + Z3, results are presented separately for the SMT solvers that support floating-points, namely CVC4 and Z3, as well as for the configuration SMT, in which all of CVC4, veriT and Z3 are used. An SMT solver is here regarded as having successfully proven a goal upon returning *unsat*, and an ATP upon reporting to have found a proof; since this integration treats SMT solvers as oracles, the provers are not required to have their proofs reconstructed internally (or even produced).

Table 2: Success rates of four prover configurations on proof goals from the complete evaluation set, by model.

	$\mathbb{U}_{16}$	$\mathbb{U}_{32}$	$\mathbb{U}_{64}$	$\mathbb{U}_{128}$	$\mathbb{U}_\text{poly}$	$\mathbb{U}_\text{some}$	$\mathbb{U}_\text{all}$	$\mathbb{U}_\text{mean}$
CVC4	17.1%	17.1%	17.1%	17.1%	18.7%	17.1%	17.1%	17.1%
Z3	17.9%	17.9%	17.9%	17.9%	18.7%	17.9%	17.9%	17.9%
SMT	17.9%	17.9%	17.9%	17.9%	18.7%	17.9%	17.9%	17.9%
ALL	18.7%	18.7%	18.7%	18.7%	**18.7%**	**18.7%**	**18.7%**	**18.7%**

	$\mathbb{I}_{16}$	$\mathbb{I}_{32}$	$\mathbb{I}_{64}$	$\mathbb{I}_{128}$		$\mathbb{I}_\text{some}$	$\mathbb{I}_\text{all}$	$\mathbb{I}_\text{mean}$
CVC4	85.4%	86.2%	82.1%	81.3%		86.2%	80.5%	83.7%
Z3	17.9%	17.9%	17.9%	17.9%		17.9%	17.9%	17.9%
SMT	85.4%	86.2%	82.1%	81.3%		86.2%	80.5%	83.7%
ALL	85.4%	86.2%	82.1%	81.3%		**86.2%**	**80.5%**	**83.7%**

Table 3: Success rates of four prover configurations on proof goals from the reduced evaluation set, by model.

	$\mathbb{U}_{16}$	$\mathbb{U}_{32}$	$\mathbb{U}_{64}$	$\mathbb{U}_{128}$	$\mathbb{U}_{\text{poly}}$	$\mathbb{U}_{\text{some}}$	$\mathbb{U}_{\text{all}}$	$\mathbb{U}_{\text{mean}}$
CVC4	26.5%	26.5%	27.9%	27.9%	29.4%	27.9%	26.5%	27.2%
Z3	32.4%	32.4%	32.4%	32.4%	32.4%	32.4%	32.4%	32.4%
SMT	33.8%	33.8%	33.8%	33.8%	33.8%	33.8%	33.8%	33.8%
ALL	36.8%	33.8%	33.8%	33.8%	**48.5%**	**36.8%**	**33.8%**	**34.6%**

	$\mathbb{I}_{16}$	$\mathbb{I}_{32}$	$\mathbb{I}_{64}$	$\mathbb{I}_{128}$		$\mathbb{I}_{\text{some}}$	$\mathbb{I}_{\text{all}}$	$\mathbb{I}_{\text{mean}}$
CVC4	83.8%	83.8%	76.5%	75.0%		83.8%	75.0%	79.8%
Z3	11.8%	11.8%	11.8%	11.8%		11.8%	11.8%	11.8%
SMT	83.8%	83.8%	76.5%	75.0%		83.8%	75.0%	79.8%
ALL	83.8%	83.8%	76.5%	76.5%		**83.8%**	**76.5%**	**80.1%**

When running Sledgehammer on the 123 formulas in the evaluation set, its relevancy filter did not have access to many useful facts other than those automatically derived from definitions. Recognizing that both ATPs and SMT solvers are heavily reliant on additional facts in an uninterpreted setting, the same models were also applied to a *reduced* evaluation set, consisting of all formulas of the original evaluation set except the first 55, which were instead made available to Sledgehammer's relevancy filter. These 55 formulas are partially designed to be used as lemmas in other proofs, and would realistically have been made available to Sledgehammer in a typical user setting. The success rates for the four prover configurations on the reduced evaluation set are shown in Table 3.

4.2 Discussion

Based on the results of our evaluation, we put forward the following observations:

Providing Z3 with an interpreted translation does more harm than good, but this is greatly outweighed by the vast increase in performance it brings to CVC4.

Not only does Z3 fail to contribute anything when working together with CVC4, but its performance is actually worse compared to when the interpretation is inactive, in terms of success rates. A big part of this can likely be attributed to the presence of bugs in Z3 (see the discussion on crashes below). CVC4, on the other hand, performs above expectations. It alone is the reason for the performance boost that is clearly visible regardless of evaluation set and measuring model. On both evaluation sets, the success rate of CVC4 is increased to around 80% when enabling the interpretation, which clearly has a strong influence over the collective performance of the default provers.

In the absence of helpful facts available to its relevancy filter, Sledgehammer's success rate improves remarkably when the translation makes use of the interpretation.

On the original evaluation set, the success rate for the default provers is almost single-handedly determined by the model applied to CVC4. Applying any of the interpreting models to the SMT solvers results in success rates over 80%, compared to the less than 20% achieved by the non-interpreting models, without the provers losing the ability to prove any goals. This performance increase demonstrates the possible gains in situations where a library of relevant basic lemmas (other than definitions) is unavailable to Sledgehammer.

Even when Sledgehammer has access to many relevant facts, the increase in performance due to interpreting floating-point types and constants, is significant.

On the reduced evaluation set, where Sledgehammer has access to more useful facts, the outcome is slightly more nuanced than on the original evaluation set. For the interpreting models, the additional facts do not seem to make much of a difference—in fact, the outcome for CVC4 and Z3 is exactly the same on the reduced evaluation set as on the corresponding goals on the original evaluation set (the lower success rates are merely a result of the removed goals being, on average, easier). The performance of the SMT solvers in the *uninterpreted* setting, however, is noticeably improved, as is that of the ATPs. Even so, when all provers work together, they manage to find proofs to an additional 42.6% of the goals with the $\mathbb{I}_{\text{all}}$ model compared to $\mathbb{U}_{\text{all}}$, increasing the success rate from 33.8% to 76.5% without failing to prove any goals proved in $\mathbb{U}_{\text{all}}$. Interestingly, the ATPs seem to perform slightly better when the floating-point types are polymorphic than when they are of a fixed format (as do the SMT solvers if the translation is uninterpreted). As a result, the performance increase from 48.5% achieved by the ALL prover configuration in $\mathbb{U}_{\text{poly}}$ to the 76.5% achieved in $\mathbb{I}_{\text{all}}$ on the reduced evaluation set comes at the cost of a few unproved goals.

The SMT solvers are more prone to errors when interpretation is enabled.

In addition to being far less likely than CVC4 to prove a formula involving floating-points, Z3 is far more unreliable; crashes are frequent, and clearly unsatisfiable formulas are sometimes reported to be satisfiable. CVC4, however, is not without its crashes either. It is not completely clear what causes the solvers to crash—it is difficult to discern a pattern, and no crash could be traced back to errors in our implementation. Similar issues were present when SMT solvers were first integrated with Sledgehammer, and were then due to bugs in the proof generation procedures of the SMT solvers [9].

Using interpretation, the SMT solvers achieve a higher success rate on formulas that lack trivial proofs in Isabelle.

Several of the formulas in the evaluation set are already provable via Isabelle's automatic methods. In a few cases, these automatic methods are simple calls without arguments, and this translation extension does not contribute much apart from saving the user the few seconds it may take to find the right method. More often, however, the automatic method requires user-provided arguments, e.g., in the form of a few facts. With the enhanced translation, Sledgehammer has a high success rate also on such formulas, eradicating the need to find (let alone prove) the required facts. The true gains of interpreting floating-point types and constants show on the formulas that were not previously provable via automatic methods; as an example, the property $x + (-y) \doteq x - y$ (where x, y, and $x - y$ are finite, and $\doteq$ denotes floating-point equality) is proved by CVC4 for all four evaluated formats within a couple of seconds when interpretation is enabled, to be compared with its original 17-line proof, involving a number of explicitly provided facts.

5 Related Work

The practice of employing automatic provers as back-ends in interactive theorem provers is not unique to Isabelle. Generic proof-delegation tools similar to Sledgehammer have also been developed for other proof assistants, e.g., Miz𝔸ℝ [40] for Mizar [2], and HOL(y)Hammer [25] for HOL Light [21] and HOL4 [37]. There are also proof-delegation tools aimed specifically toward SMT solvers, e.g., Smtlink [33] for ACL2 [26] and SMTCoq [1] for Coq [7].

Single integrations of SMT solvers have perhaps been more common than these larger-scale tools. The interactive theorem prover PVS [31] is tightly connected with the SMT solver Yices [17] (and its predecessor ICS), which has been available as a decision procedure for a long time. An oracle integration of Yices in Isabelle by Erkök and Matthews [19] makes use of its dedicated decision procedures, but refrains from translating into SMT-LIB, and instead targets the native input format of Yices due to its expressiveness. Weber [41] proposes a similar oracle integration of Yices into HOL4, but extends it with support for additional SMT solvers via the SMT-LIB format. This integration has since been supplemented with proof reconstruction and become part of HOL(y)Hammer [11].

The work presented herein is based on the original integration of SMT solvers in Isabelle by Blanchette et al. [9]. It is dependent on all aspects of their translation into SMT-LIB, including the interpretation of bit vector types and constants. In this sense, it also bears resemblance to how SMTCoq was recently extended with dedicated support for the theory of bit vectors [18].

Despite readily available formalizations of IEEE 754 floating-point arithmetic, that also have been used to verify various floating-point related properties, e.g., in HOL Light [22], ACL2 [35], and Coq [15], no integration of SMT solvers in ITPs seems to make use of the former's dedicated decision procedures of floating-points at present. Superficially, the work perhaps most similar to ours is a somewhat recent Why3 [10] formalization of floating-point arithmetic and its mapping to the SMT-LIB floating-point theory [20]. Why3, however, is not a prover itself, but merely a stand-alone proof-delegation tool relying completely on external provers, and thus the higher automation of interactive theorem provers is not a shared objective.

6 Conclusions and Future Work

In the years since its introduction to Isabelle, Sledgehammer has seen a number of improvements. In varying degree, they have each gradually brought us closer toward the ultimate goal of full proof automation in interactive theorem provers. In light of this ambitious goal, our contribution may seem small, but it is, as our evaluation in Section 4 shows, an unmistakable step in the right direction.

By enhancing the translation from Isabelle to SMT-LIB with the interpretation of floating-point types and constants described in Section 3, Sledgehammer (when treating SMT solvers as oracles) sees a significant increase in success rates for proof obligations involving such types and constants. Our integration does not require formulas to be completely interpretable in the SMT-LIB floating-point logic for the improvements to apply; the SMT solvers are instructed to reason within a more general logic where uninterpreted and interpreted sorts and functions are combined freely. Most importantly, however, the improvements apply to goals that were previously not provable by any automatic method in Isabelle, showing that enhancing the translation with floating-point interpretation does increase proof automation. The increase in performance now brought to Sledgehammer means that more proofs can be carried out automatically, relieving the user of this burden and saving in on the time and effort spent constructing proofs manually.

There are two notable limitations on the translation of floating-point types: it is restricted to fixed formats, and it is presently unsound. The restriction to fixed formats, stemming from the lack of support for polymorphism in version 2 of SMT-LIB, is in many cases just a mild inconvenience—polymorphic floating-points are left uninterpreted, and floating-point formats of interest need to be treated separately for Sledgehammer to enable the enhanced translation. The unsoundness, however, may result in Sledgehammer proving "false theorems" if the user is not careful.

Although our integration does achieve an automation boost in and of itself, it is worth reiterating that it is only the first step in the larger project of (securely) increasing the automation of floating-point reasoning in Isabelle. What follows is a discussion on the possible ways forward with respect to this objective.

Redefining the Isabelle/HOL floating-point type. The perhaps most pressing matter is to eliminate the translation's source of unsoundness. This means that the Isabelle/HOL definition of the floating-point type needs to be harmonized with that of the SMT-LIB sort.

The change itself is simple enough: instead of defining the type via an isomorphism from the set $\mathbf{BV}_1 \times \mathbf{BV}_{\mathtt{m}} \times \mathbf{BV}_{\mathtt{n}}$, a set like

$$\mathbf{A} \coloneqq \mathbf{Z}_{\mathtt{m},\mathtt{n}} \cup \mathbf{S}_{\mathtt{m},\mathtt{n}} \cup \mathbf{N}_{\mathtt{m},\mathtt{n}} \cup \mathbf{I}_{\mathtt{m},\mathtt{n}} \cup \{(0, \mathbf{1}_{\mathtt{m}}, \mathbf{1}_{\mathtt{n}})\}$$

could be used without interfering with the overall structure of the theory. A new function $f \colon \mathbf{BV}_1 \times \mathbf{BV}_{\mathtt{m}} \times \mathbf{BV}_{\mathtt{n}} \to \mathbf{A}$ such that

$$f(\mathtt{s},\mathtt{e},\mathtt{f}) = \begin{cases} (0, \mathbf{1}_{\mathtt{m}}, \mathbf{1}_{\mathtt{n}}) & \text{if } \mathtt{e} = \mathbf{1}_{\mathtt{m}} \text{ and } \mathtt{f} \neq \mathbf{0}_{\mathtt{n}} \\ (\mathtt{s},\mathtt{e},\mathtt{f}) & \text{otherwise} \end{cases}$$

would then replace `Abs_float` as the function that corresponds to the SMT-LIB function `fp`. The challenge in this case lies in updating Isabelle theories whose contents depend on the present definition of the floating-point type.

Improving SMT solver integrations in Isabelle. As evident from the evaluation in Section 4, the integration of SMT solvers in Isabelle leaves much to be desired from the perspective of floating-point reasoning—only CVC4 reaches acceptable success rates. The overall disappointing display by Z3 is most likely due to version 4.4.0 being outdated, so to not waste the potential of Z3, a version upgrade is necessary. Whereas Isabelle2021 includes up-to-date versions of CVC4 and veriT, the support for newer versions of Z3 is very limited; Z3 holds a special place among the SMT solvers integrated into Isabelle, as it is the only one having its proofs reconstructed. Relying on the proof format of the solver, the reconstruction process is version dependent, since the decision procedures are under constant improvement [8]. Thus the upgrade is more than just a matter of swapping out the executable. The typically superior performance of Z3 (along with preliminary manual experiments that we performed) indicates that the payoff could be well worth the trouble. An integration of currently unsupported SMT solvers with dedicated support for floating-points (e.g., MathSAT [14]) should also be considered.

Adjusting for SMT-LIB 3. Led by the ambitious Matryoshka project[7], there is ongoing work in obtaining a tighter integration of automatic provers, including SMT solvers, with proof assistants. One of the means by which to achieve this is via support for higher-order logic in

[7]https://matryoshka-project.github.io/

these provers [4]. Most likely, SMT-LIB 3—the next major update to SMT-LIB—will facilitate these changes by adding support for polymorphism [3]. In order to utilize these advancements, the translation module for floating-points needs to be updated accordingly.

Reconstructing proofs. Integrating external provers as oracles, as done in this paper, puts complete trust into the correctness of both the provers and the translation to their input format. Compared to Isabelle, the trusted code base in SMT solvers is large and much more likely to contain bugs. Proofs found by an SMT solver integrated as an oracle circumvent Isabelle's small inference kernel, and are therefore less trustworthy. The approach preferred by the interactive theorem proving community is that of a skeptic [23]—external proofs should be reconstructed internally. If successful, this approach combines the speed of the SMT solver with the reliability of the proof assistant.

By using proof details provided by external solvers, Isabelle is already able to reconstruct many of their proofs. This ability, however, is limited and does not handle reasoning specific to certain SMT-LIB theories. Efficient reconstruction of proofs that use floating-point reasoning requires both improving on the provided proof information, and translating theory-specific inferences. Implementing efficient proof methods in Isabelle/HOL is also a part of this challenge. A decision procedure for floating-point arithmetic implemented on top of Isabelle's inference kernel would both facilitate the reconstruction of proofs and increase the overall automation of internal floating-point reasoning.

Both the implementation of proof reconstruction and a decision procedure for floating-points in Isabelle will require substantial work. The promising results in this paper are reassuring in that the effort would not be wasted.

7 Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs, First International Conference – CPP 2011*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.

[2] G. Bancerek, C. Bylinski, A. Grabowski, A. Kornilowicz, R. Matuszewski, A. Naumowicz, K. Pak, and J. Urban. Mizar: State-of-the-art and beyond. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Intelligent Computer Mathematics, International Conference – CICM 2015*, volume 9150 of *Lecture Notes in Computer Science*, pages 261–279. Springer, 2015.

[3] H. Barbosa, J. C. Blanchette, S. Cruanes, D. E. Ouraoui, and P. Fontaine. Language and proofs for higher-order SMT (work in progress). In C. Dubois and B. W. Paleo, editors, *Fifth Workshop on Proof eXchange for Theorem Proving – PxTP 2017*, volume 262 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–22, 2017.

[4] H. Barbosa, A. Reynolds, D. E. Ouraoui, C. Tinelli, and C. W. Barrett. Extending SMT solvers to higher-order logic. In P. Fontaine, editor, *27th International Conference on Automated Deduction – CADE 27*, volume 11716 of *Lecture Notes in Computer Science*, pages 35–54. Springer, 2019.

[5] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.smt-lib.org`.

[6] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification, 23rd International Conference – CAV 2011*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[7] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[8] N. Bjørner and L. Nachmanson. Navigating the universe of Z3 theory solvers. In G. Carvalho and V. Stolz, editors, *Formal Methods: Foundations and Applications, 23rd Brazilian Symposium – SBMF 2020*, volume 12475 of *Lecture Notes in Computer Science*, pages 8–24. Springer, 2020.

[9] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.

[10] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[11] S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving, First International Conference – ITP 2010*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010.

[12] T. Bouton, D. C. B. D. Oliveira, D. Déharbe, and P. Fontaine. veriT: An open, trustable and efficient SMT-solver. In R. A. Schmidt, editor, *22nd International Conference on Automated Deduction – CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[13] M. Brain, C. Tinelli, P. Rümmer, and T. Wahl. An automatable formal semantics for IEEE-754 floating-point arithmetic. In *22nd IEEE Symposium on Computer Arithmetic – ARITH 2015*, pages 160–167. IEEE, 2015.

[14] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 19th International Conference – TACAS 2013*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

[15] M. Daumas and S. Boldo. Necessary and sufficient conditions for exact floating point operations. Research Report RR-4644, LIP RR-2002-44, INRIA, LIP, 2002.

[16] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference – TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[17] B. Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer Aided Verification, 26th International Conference – CAV 2014*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.

[18] B. Ekici, G. Katz, C. Keller, A. Mebsout, A. J. Reynolds, and C. Tinelli. Extending SMTCoq, a certified checker for SMT (extended abstract). In J. C. Blanchette and C. Kaliszyk, editors, *First International Workshop on Hammers for Type Theories – HaTT@IJCAR 2016*, volume 210 of *Electronic Proceedings in Theoretical Computer Science*, pages 21–29, 2016.

[19] L. Erkök and J. Matthews. Using Yices as an automated solver in Isabelle/HOL. In J. Rushby and N. Shankar, editors, *AFM'08: Third Workshop on Automated Formal Methods*, pages 3–13, 2008.

[20] C. Fumex, C. Marché, and Y. Moy. Automated verification of floating-point computations in Ada programs. Research Report RR-9060, Inria Saclay Ile de France, 2017.

[21] J. Harrison. HOL Light: A tutorial introduction. In M. K. Srivas and A. J. Camilleri, editors, *Formal Methods in Computer-Aided Design, First International Conference – FMCAD '96*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[22] J. Harrison. Floating-point verification using theorem proving. In M. Bernardo and A. Cimatti, editors, *Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems – SFM 2006*, volume 3965 of *Lecture Notes in Computer Science*, pages 211–242. Springer, 2006.

[23] J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21(3):279–294, 1998.

[24] IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[25] C. Kaliszyk and J. Urban. HOL(y)Hammer: Online ATP service for HOL Light. *Mathematics in Computer Science*, 9(1):5–22, 2015.

[26] M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.

[27] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[28] J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009.

[29] J.-M. Muller, N. Brisebarre, F. de Dinechin, C. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[30] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[31] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction – CADE 11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[32] L. C. Paulson and J. C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In G. Sutcliffe, S. Schulz, and E. Ternovska, editors, *The 8th International Workshop on the Implementation of Logics – IWIL 2010*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010.

[33] Y. Peng and M. R. Greenstreet. Extending ACL2 with SMT solvers. In M. Kaufmann and D. L. Rager, editors, *Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications – ACL2 2015*, volume 192 of *Electronic Proceedings in Theoretical Computer Science*, pages 61–77, 2015.

[34] A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2-3):91–110, 2002.

[35] D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[36] S. Schulz, S. Cruanes, and P. Vukmirovic. Faster, higher, stronger: E 2.3. In P. Fontaine, editor, *27th International Conference on Automated Deduction – CADE 27*, volume 11716 of *Lecture Notes in Computer Science*, pages 495–507. Springer, 2019.

[37] K. Slind and M. Norrish. A brief overview of HOL4. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference – TPHOLs 2008*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008.

[38] P. H. Sterbenz. *Floating-point computation*. Prentice-Hall, 1974.

[39] O. Torstensson. Oracle integration of floating-point solvers with Isabelle. Master's thesis, Uppsala University, Department of Information Technology, 2021.

[40] J. Urban, P. Rudnicki, and G. Sutcliffe. ATP and presentation service for Mizar formalizations. *Journal of Automated Reasoning*, 50(2):229–241, 2013.

[41] T. Weber. SMT solvers: New oracles for the HOL theorem prover. *International Journal on*

Software Tools for Technology Transfer, 13(5):419–429, 2011.

[42] C. Weidenbach. Combining superposition, sorts and splitting. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1965–2013. Elsevier and MIT Press, 2001.

[43] L. Yu. A formal model of IEEE floating point arithmetic. *Archive of Formal Proofs*, 2013. http://isa-afp.org/entries/IEEE_Floating_Point.html, Formal proof development.