Formal Verification of the IEEE P3109 Standard for Binary Floating-point Formats for Machine Learning

Christoph M. Wintersteiger
Imandra Inc
christoph@imandra.ai

***Abstract*—We present a formalization of an upcoming standard for floating-point formats for machine learning by the IEEE P3109 working group. This includes a definition of a number of small (< 16 bit) formats and a specification of arithmetic functions that operate on such numbers, as well as format conversion function, including conversions to and from IEEE 754 formats. We report on our experience with the use of an automated theorem prover for verification and analysis of our formalization of the specification, and on the utility of the formalization in future implementations of P3109-compliant hardware and software.**

1. Introduction

Progress in machine learning, and artificial intelligence in general, has been tremendous and it still continues at rapid speed. The arithmetic requirements for these systems are significantly different from other domains, for instance the precision of such operations is often much lower. Multiple generations of new hardware designs that incorporate low-precision floating-point cores are now in practical use. With different points of focus for each hardware developer, those floating-point cores do not have a common semantics however, and it can be challenging to port an application from one platform to another. This motivates the design of a new standard that is broad enough to cover the requirements of most applications, but at the same time provides a common semantics that improves portability and, in general, makes them more predictable. The IEEE working group P3109 [1] is currently working on such a standard and their latest draft report [2] defines all of the operations intended to be included, with some work on the precision and rounding of operations still in progress.

Floating-point numbers have been of interest to the formal verification community for a long time. One of the early efforts was by Barrett, who developed a formalization of IEEE 754 in Z-notation [3] as did Harrison in HOL light [4], [5]. Recently, Yu [6] started porting Harrison's formalization to Isabelle proof assistant [7], and an effort to develop a formalization in Coq led by Boldo and Melquiond [8]. Today, Satisfiability Modulo Theories (SMT) solvers [9] are a crucial component in theorem provers and program verifiers. Their standard set of supported theories now includes support for a theory of floating-point numbers that is close to IEEE 754, but that is parametrizable by bit-widths for exponents and significands.

2. Background

P3109 defines formats for 3 to 15-bit floating-point numbers, each with a significand precision of one up to $n-1$ bits (including a hidden significand bit). P3109 numbers include positive and negative infinities as well as a single NaN without payload and a single zero. It further specifies the usual arithmetic operations, square roots, and natural and binary logarithms and exponentials. Additionally it includes versions of addition and multiplication with (log-scale) scaling factors and a fused-multiply-add operation that takes a (scaled) IEEE 754 value as a summand alongside P3109 values and produces an IEEE 754 result. For more details on these operations, see the interim report of the working group [2].

Our work is based on ImandraX, the latest generation of theorem provers and program verifiers from Imandra [10]. It accepts input in the Imandra Modelling Language (IML), a subset of OCaml with minor modifications and additions. IML is a strongly-typed functional language that supports polymorphic higher-order recursive functions, but does not support exceptions or assertions, such that functions are always total. (It being a dialect of ML, note that `(f x y)` represents a function call equivalent to `f(x, y)` in other languages.) It does not feature automatic type conversions or operator overloading. Theorems are stated in terms of input variables that are implicitly universally quantified, but the language does not support explicit quantifiers or alternation. This is by design, to keep verification problems and code complexity on a lower level (complexity-theoretic and otherwise), while supporting just enough to tackle most problems encountered in industrial applications.

The type `int` represents mathematical (unbounded) integers and a `real` represents a mathematical (non-approximated) real. We use extended reals, i.e. reals with the addition of infinity (but not NaN), for which the type is `ExReal.t`. Operations on extended reals require us to exclude some combinations of inputs. For instance, the result of an addition of a positive and a negative infinity is undefined. One of the goals of our formalization is to enable us to prove formally that the P3109 specification never depends on any such undefined cases, which it often

2576-2265/25/$31.00 ©2025 IEEE
DOI 10.1109/ARITH64983.2025.00032

achieves by explicit specification of special cases before the operation on extended reals is used. Since the language does not support operator overloading, we add a dot to the arithmetic operators one extended reals, e.g. `+.` for addition.

Types are usually defined within their own modules, using the module name to identify the type in question. For instance, we use the `Result.t` type to track success and errors:

```
module Result = struct
  type ( a,  b) t = Ok of  a | Error of  b
end
```

This means, a function returning a `Result.t`, either returns an `Ok` with payload of type `a`, or an `Error` of type `b`. (Here, we only use errors of type `string`.)

On the theorem proving and verification side, ImandraX has many advanced features like a symbolic model checker, automated induction, a powerful simplifier and symbolic execution engine with lemma-based conditional rewriting and forward-chaining, composable tactics, counterexamples, state-space decompositions, and decision procedures for various theories.

SMT solvers like Z3 [11], CVC5 [12], and MathSAT [13] support the SMT-LIB theory of floating-point numbers, which is largely based on IEEE 754-2008, but with support for custom exponent and significand widths and a single NaN without payload. Many modern program verifiers employ SMT solvers to discharge first-order verification conditions as they often perform well out of the box or with minor adjustments. It is important to note that ImandraX also uses Z3, but our formalization of P3109 does not make use of the SMT-LIB theory of floating-point numbers, instead it is based on (mathematical) integers and (extended) reals.

3. Formalization

Our formalization starts with the definition of a type for P3109 numbers, for which we use (unbounded) integers to support varying bit-widths:

```
type t = int
```

For n-bit P3109 numbers, integers outside of $[0, 2^n - 1]$ are rejected during decoding (producing an `Error`).

The different P3109 formats are simply an enumeration of constants:

```
module Format = struct
  type t = B3P1 | ... | B8P2 | B8P3 | B8P4 |
       B8P5 | B8P6 | B8P7 | ... | B15P14
  ...
end
```

where `p` in `BnPp` indicates the precision of the format (i.e. the number of significand bits, including the hidden one).

Projection specifications are combinations of saturation and rounding modes (Listing 1). and the function `project` applies saturation and rounding according to such projection specifications.

```
module SaturationMode = struct
  type t = bool
end

module RoundingMode = struct
  type t = TowardZero | TowardNegative |
      TowardPositive | NearestTiesToEven |
      NearestTiesToAway
end

module Projection = struct
  type t = SaturationMode.t * RoundingMode.t
end
```

Listing 1. Projections

With this, we are now ready to investigate an example of the definition of scaled addition. Figure 1 shows the textual definition in the standard documents, while Listing 2 shows our formalization thereof. The operation takes two P3109 values (x and y) as well as two log-scale (integer) scaling factors s_x and s_y. P3109 leaves the choice of supported ranges for s_x and s_y to the implementation, so that for our formalization we assume that they are unbounded integers, such as to cover all compliant choices of ranges.

The specification of the behavior of operations is by pattern matching with the first case in top-down order that applies defining the behavior ($*$ means "anything"). In this case, the operation returns a NaN when either of the inputs is a NaN, or when they are infinities of opposite signs. Otherwise, the result is $X \times 2^{s_x} + Y \times 2^{s_y}$, where X and Y are the decoded P3109 values, i.e. their equivalent in the extended reals (still including infinities of equal signs), computed by `decode`. The result is then saturated and rounded by the function Project. If any of the intermediate functions produces an `Error`, it is propagated to the outside, i.e. the return value of `internal_add_scaled` would then be an `Error` too.

Of course, the actual (textual) specification always returns a P3109 value (z) and not a `Result.t`. We therefore define the (external) formal specification of `add_scaled` by propagating successful results and, for now, by turning errors into NaNs, as shown in Listing 3. (Note that the return type is now `Float8.t`.) We know however, that this error case is in fact unreachable, and we can show this by proving the theorem given in Listing 4, which asserts that the result of `internal_add_scaled` is always an `Ok`. The proof of this theorem is relatively short and essentially consists of a path analysis: first we prove that `decode` and `project` never return errors (similarly to the theorem in Listing 4), which implies that the last two `Error` cases are unreachable, as well as the path through `project`, leaving only the potential error as a result of evaluation of the arithmetic term on extended reals. Since `*.`, `+.`, and `^.` are defined on all combinations of infinities possibly remaining after the special case handling at the beginning of the function (mainly $\infty + \infty$ and $-\infty + -\infty$ and a potential $\pm\infty \times 0$), this is also relatively easy to show. ImandraX finds these proofs

Scaled addition

Compute $X \times 2^{s_x} + Y \times 2^{s_y}$, and return a P3109 value. Scaling is applied in the extended reals, before projection to the target format.

Signature

$\text{AddScaled}_{f_x,f_y,f_z,\pi}(x, s_x, y, s_y) \to z$

Parameters

f_x : format of x
f_y : format of y
f_z : format of z
π : projection specification

Operands

x : P3109 value, format f_x
s_x : integer log-scale factor for x
y : P3109 value, format f_y
s_y : integer log-scale factor for y

Output

z : P3109 value, format f_z

Behavior

AddScaled(NaN, $*$, $*$, $*$) $\to$ NaN
AddScaled($*$, $*$, NaN, $*$) $\to$ NaN
AddScaled(-Inf, $*$, Inf, $*$) $\to$ NaN
AddScaled(Inf, $*$, -Inf, $*$) $\to$ NaN
AddScaled(x, s_x, y, s_y) $\to \text{Project}_{f_z,\pi}(Z)$, where
$Z = X \times 2^{s_x} + Y \times 2^{s_y}$
$X = \text{Decode}_{f_x}(x)$
$Y = \text{Decode}_{f_y}(y)$

Figure 1. Textual specification of scaled addition

```
let internal_add_scaled
  (f_x : Format.t) (f_y : Format.t)
  (f_z : Format.t) (pi : Projection.t)
  (x : Float8.t) (s_x : int)
  (y : Float8.t) (s_y : int)
  : (t, string) Result.t =
  let open NaNOrExReal in
  match x, y with
  | _, y when y = nan -> Ok nan
  | x, _ when x = nan -> Ok nan
  | x, y when x = ninf && y = pinf -> Ok nan
  | x, y when x = pinf && y = ninf -> Ok nan
  | x, y ->
    let x = decode f_x x in
    let y = decode f_y y in
    (match x, y with
    | Ok NaN, _ |  _, Ok NaN -> Ok nan
    | Ok (XR x) , Ok (XR y) ->
      let open ExReal.ResultInfix in
      (match ((Ok x) *. (2 ^. s_x)) +.
             ((Ok y) *. (2 ^. s_y)) with
      | Ok z -> project f_z pi z
      | Error e -> Error e)
    | _, Error e -> Error e
    | Error e, _ -> Error e)
```

Listing 2. Formal specification of scaled addition (internal)

```
let add_scaled
  (f_x : Format.t) (f_y : Format.t)
  (f_z : Format.t) (pi : Projection.t)
  (x : Float8.t) (s_x : int)
  (y : Float8.t) (s_y : int)
  : Float8.t =
  match
    internal_add_scaled
      f_x f_y f_z pi x s_x y s_y
  with
  | Ok x -> x
  | Error _ -> nan (* unreachable by theorem
       internal_add_scaled_ok *)
```

Listing 3. Formal specification of scaled addition (external)

```
theorem internal_add_scaled_ok
  (f_x : Format.t) (f_y : Format.t)
  (f_z : Format.t) (pi : Projection.t)
  (x : Float8.t) (s_x : int)
  (y : Float8.t) (s_y : int) =
  Result.is_ok (Float8.internal_add_scaled
      f_x f_y f_z pi x s_x y s_y)
```

Listing 4. Theorem stating that `internal_add_scaled` always returns an `Ok`

fully automatically once given the right theorems to prove.

Some of the intermediate theorems that allow us to get short proofs for the top-level theorems are of different character. For instance, the function `project` (Listing 5) employs within it the function `encode`, which encodes an extended real number into a P3109 number. It only succeeds if the input is within the range of the respective floating-point format (within smallest and largest representable numbers). To guarantee that `project` never fails, we thus prove that `saturate` produces results that are indeed always within the range of the format. Once equipped with such a generic result, many of the other proofs become significantly shorter as they can simply re-apply the theorem instead of performing costly numerical analysis.

In total, the formalization of P3109 is about 3 kloc containing 28 theorems of the form above, 34 intermediate lemmas, 121 theorems about format ranges, 83 termination proofs, and 91 test cases (for a total of 357 proof obligations), which ImandraX discharges in 23 minutes of CPU

```
let project
  (f : Format.t) (pi : Projection.t)
  (x : ExReal.t)
  : (t, string) Result.t =
  let _, p, b, m = Format.
      get_format_parameters f in
  let sat, rnd = pi in
  let r : ExReal.t = round_to_precision p b
      rnd x in
  let s : ExReal.t = saturate m sat rnd r in
  encode f (NaNOrExReal.XR s)
```

Listing 5. Formal specification of projection

time on an Intel i9-12900KF.

4. Additional benefits

Our primary motivation for developing a formal version of the P3109 specification is to find inconsistencies within the standard itself, while the standard is still being developed. This gives us confidence that the standard itself specifies all of the necessary properties of formats and operations and that it is indeed implementable in practice. While a significant amount of that can arguably be achieved with weaker methods, a full formalization comes with additional benefits:

- Executable specification: From the formal specification and proofs of its properties, we can automatically extract fully executable code in very little time. The generated code is guaranteed to have the properties that are proven on the specification and therefore allows us to easily investigate new properties or the general behavior of all operations. Since it is essentially a complete implementation of the standard in software, it is very easy to use as a test-case generator or general test reference for other implementations of the standard. The only minor downside is that it is sometimes inefficient, relative to targeted and optimized implementations. In our case, ImandraX automatically generates OCaml code that is standard-compliant by construction, and immediately usable by other OCaml programs.
- Formal reference: A formal specification also serves as a reference for other formalizations. For instance, we can design a more performant version of an operation and then prove that it has the same semantics as the original one. Take, for instance, P3109 negation, which is specified as the decoding of a number into extended reals, negation performed on the extended reals, and the result being encoded into a P3109 value again. This procedure is not efficient in practice; we really just want to flip the sign bit (except for NaNs and zeroes, see Listing 6 for an implementation specialized to 8-bit formats). This is trivial to specify and, in this simple case, also easy to prove correct.

5. Conclusion

We present a formalization of the upcoming IEEE P3109 standard for floating-point formats for machine learning, developed during formation of the standard, providing us with increased confidence that our specification is consistent and implementable. It is publicly available for immediate reuse by others interested in the properties of P3109 numbers (see [14]) and it is supplied in a format that is easily re-used in other analysis tools, many of which are based on similar input languages. While the theorem prover we employ is an industrial product, the formalization itself does not depend on proprietary features and other provers may require only minor changes to proof strategies to find similar proofs.

```
let negate (f : Format.t) (x : t) : Float8.t
    =
  if is_nan f x || is_zero f x then x
  else
  {
    b7 = not x.b7; b6 = x.b6; b5 = x.b5;
    b4 = x.b4;     b3 = x.b3; b2 = x.b2;
    b1 = x.b1;     b0 = x.b0;
  }

theorem crct_neg (f:Format.t) (x:Float8.t) =
  Format.k f = 8 ==>
  i2s (negate f x) = Specification.Float8.
      negate f (i2s x)
```

Listing 6. Efficient implementation of negation

While not the primary purpose of our formalization, it also allows us to extract a standard-compliant implementation of the formalization in software, which is easy to use as a test case generator or test reference for other implementations in hardware or software.

Future work includes a variety of proofs about the precision of irrational and transcendental functions in P3109.

References

[1] [Online]. Available: https://standards.ieee.org/ieee/3109/11165

[2] [Online]. Available: https://github.com/P3109/Public

[3] G. Barrett, "Formal methods applied to a floating-point number system," *IEEE Trans. Softw. Eng.*, vol. 15, no. 5, May 1989.

[4] J. Harrison, "A machine-checked theory of floating point arithmetic," in *Theorem Proving in Higher Order Logics*. Springer, 1999.

[5] ——, "HOL Light: An overview," in *Proc. Conf. on Theorem Proving in Higher Order Logics TPHOLs*, ser. LNCS, vol. 5674. Springer, 2009.

[6] L. Yu, "A Formal Model of IEEE Floating Point Arithmetic," *Archive of Formal Proofs*, May 2024. [Online]. Available: https://www.isa-afp.org/browser_info/current/AFP/IEEE_Floating_Point/document.pdf

[7] L. C. Paulson, "Natural deduction as higher-order resolution," *J. Logic Programming*, vol. 3, no. 3, 1986.

[8] S. Boldo and G. Melquiond, *Computer Arithmetic and Formal Proofs*. Elsevier, 2017.

[9] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017. [Online]. Available: www.SMT-LIB.org

[10] [Online]. Available: https://imandra.ai

[11] L. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proc. Conf. Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[12] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Proc. Conf. Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, ser. LNCS, vol. 13243. Springer, 2022.

[13] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT solver," in *Proc. Conf. Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, ser. LNCS, vol. 7795. Springer, 2013.

[14] [Online]. Available: https://github.com/imandra-ai/ieee-p3109