Dietrich Geisler

Implementation of a Floating-Point Type in Boogie

**Abstract**

The objective of this project is to implement a floating point type into the Boogie language to improve the versatility of the Boogie and SMACK verification tools. This new type will allow for the IEEE standard of 32-bit floating points values, with future modifications to allow for the IEEE 64-bit value and non-standard n-bit floating point values with a provided mantissa and exponent size. The new float type in Boogie follows standard Boogie syntax with a type name of float and a constant declaration keyword fp. For more details on float declaration syntax in Boogie, see section 2.

**Background**

The floating point (fp) is used as the standard for real number representation in binary. The fp representation of some real number r consists of an exponent e and mantissa m such that:

Where e is an integer and m is a real value less than 2. This representation is similar to scientific notation, such as how 2,057 = 2.057x103. As an example of floating point representation, consider the value 4.5. A binary representation of this number is 100.1, which gives us e = 2, m = 1.001 since multiplying the binary value 1.001 = 1.125 by 22 = 4 gives us the value 100.1 = 4.5. Note that this representation naturally follows for both negative-valued reals and negative exponents.

The value stored in a floating point (fp) is necessarily an approximation for the real value it represents, since the set of real values is both complete and uncountably infinite. As such, the IEEE 32-bit fp standard limits floating point approximation to 32 bits consisting of a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa. These bounds imply that the following inequalities hold:

-128 <= e <= 127 -2128 < fp value < 2128

For the purpose of discussing fp theory, it will be assumed that all fp values follow the IEEE 32-bit standard. Note, however, that the principles presented are also applicable to any fp size.

The precision of fp approximation naturally depends on the size of the mantissa. The precision of approximating a given real number r, however, also depends on the size of the exponent required to represent r. The precision given by an imperfect floating point approximation is entirely dependent on the strength of least significant bit of the mantissa. That is, if the least significant bit of r represents 22, for example, then the fp representation of r is within 4 integer units of the actual value of r. More specifically, the precision of an fp representation of some r can be given by 2exponent – 23, since the least significant bit of the mantissa represents a value 223 orders of magnitude more significant than the value of the exponent.

While there are standards to reduce floating point rounding error, which will not be discussed in this paper, floating point rounding in calculations has the potential to cause serious issues in code. Consider, for instance, the result of adding the decimal values 0.1 + 0.1 + 0.1. While we would intuitively consider the result of this addition to be 0.1, an unguarded floating point operation of this sort would result in a value slightly greater than 0.1 due to the inexact fp approximation of 0.1. That is, for unguarded addition where float a = 0.1; the expression a + a + a == 0.3 returns false. For more information on floating point operations, see section 3.

Despite the issues with fp approximation, floats are by necessity used throughout computing, particularly in modeling and graphics. As such, it is often necessary to verify floating point code, which is the objective of this project.

**Section 1: Overview of Boogie[[1]](#footnote-1)**

The purpose of boogie is to translate boogie code to a format readable by SMT solvers such as z3. As part of this translation, Boogie performs several steps of abstraction and simplification. First, Boogie parses the

1. Note that this section is based on reading and interpreting boogie code with minimal documentation. As such, any information here is subject to change. [↑](#footnote-ref-1)