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 tool. 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 3.

Section 1: 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:

$$r \approx m * 2^{e}$$

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 = 2.057x10^3. 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 2^2 = 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 $\qquad$ -2^{128} < fp value < 2^{128}

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 2^2, 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 $2^{\text{exponent} - 23}$, since the least significant bit of the mantissa represents a value 2^{23} 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 4.

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 2: Overview of Boogie[1]

Boogie is a verification tool created by the Microsoft Research Team for the purpose of translating 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 data from the provided bpl file. Second, as data is parsed, Boogie creates abstract types and expressions to represent the variables and expressions parsed. As part of generating expressions, Boogie performs simple checks and simplifications for trivial results; for example, the expression x == x becomes True. Third, Boogie uses the abstract expressions generated during the previous step to generate SMT verifiable expressions. Finally, Boogie generates a file for z3 to verify and reports the result. Each of the above steps is described in further detail below.

Boogie parses incoming code by individual token. Each token retrieved from the Scanner has a type which informs the Parser class how to proceed. If a token has a real type, for instance, the Parser class will create and store an instance of the real type. Each of these types and expressions is one of many kinds of abstract expressions contained in the Absy* classes. As stated above, some simple evaluations and simplifications are made to the generated abstract expressions. Finally, the generated expressions are stored in a program for conversion into verifiable Boolean expressions.

[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.

Once the program described above is generated, the VC class and the Boogie2VCExpr classes handle the conversion of abstract expressions to verifiable expressions. The method by which this translation occurs is undetermined, but it is known that the expression nodes generated during parsing are visited to create variable declarations and Boolean expressions. Once the parsed expressions are converted to SMT verifiable expressions, they are translated to z3-readable code and fed to the verifier. From here, results are generated based on z3 model output from the given expressions.

Section 3: Floating Point Syntax

The float type introduced to Boogie is a 32-bit floating point value as described above. To declare a variable x as a 32-bit fp value, simply state:

```
var x : float;
```

(As of this writing, the following feature has not been implemented). To declare a variable x as a fp value with an exponent of size e and a mantissa of size m, simply state:

```
var x : float(e m);
```

To declare a constant 32-bit floating point value, the following statements can be used, where d is a decimal to be approximated, e is the exact exponent value, m is the exact mantissa value, se is the bit size of the exponent, and sm is the bit size of the mantissa. Note that in cases where se and sm are not explicitly declared, they are assumed to be 8 and 23, respectively, as per the IEEE 32-bit floating point standard:

```
fp (d);          fp (e m);          fp (d se sm);          fp(e m se sm);
```

It should be noted that when approximating a decimal d, the float type uses the round function described in the following section.

Section 4: Floating Point Operations

The following operations are or will be implemented into Boogie for the float type as described below. Any operation marked with a star (*) has not been implemented at the time of this writing.

Equality (==) returns a Boolean based on the exponent and mantissa comparisons of two float values. The equality operation is illegal for comparisons between floats of differing exponent or mantissa sizes, for comparisons between floats and reals, and for comparisons between floats and ints.

Inequality (>, <, >=, <=) returns a Boolean based on the result of subtracting one float from the other as compared to zero. That is, if $a – b > 0$, then $a > b$. This operation follows the same laws as equality.

*Round (round(r)) creates the closest possible fp approximation to given real r. The exponent of this fp is determined by multiplying or dividing r until either the value of r is within $2^{\text{exponent} - 23}$ orders of magnitude to either 1 or -1 or the bit limit of the exponent has been exceeded. The remaining decimal r is then converted to an n-bit mantissa (where n is the allowed size of our mantissa).

*Addition (a + b) returns the float result of adding floats a and b. To do this, the mantissa of min(a, b) is shifted until the exponent values of a and b match. The resulting mantissas are then added. If the value of the resulting mantissa is not within (-2, -1)U(1, 2), the exponent is updated appropriately. Note that the implicit most significant bit of the mantissa must be included in this calculation. The result of INF + x is INF. The result of INF + -INF is NaN. The addition operation is undefined for differing mantissa sizes, differing exponent sizes, addition between a float and a real, and addition between a float and an int.

*Subtraction (a - b) returns the result of negating b and performing the addition between a and the modified b.

*Multiplication (a * b) returns the float result of multiplying floats a and b. The algorithm for this operation has not been well-researched but will be implemented. Multiplication is undefined for differing mantissa sizes, differing exponent sizes, multiplication between a float and a real, and multiplication between a float and an int.

*Division (a / b) returns the float result of dividing float b from float a. The algorithm for this operation has not been well-researched but will be implemented. Division is undefined for differing mantissa sizes, differing exponent sizes, division between a float and a real, and division between a float and an int.

Section 5: Modifications to Boogie

This section will act as a summary of the modifications that have been made to Boogie for implementation of the float type. For a complete list of all changes made as part of this project, see the github page for this project listed under references. If the flow of Boogie is unclear, reference Section 2.

First, the BigFloat class was introduced to define the float type and its operations. This class was directly modified from the preexisting BigDec class. Work on this class is ongoing, and mostly involves modifying functions to correctly model floating point behavior.

Second, the float type and fp constant were introduced to the Parser and Scanner classes. Note that the token values of 97 and 98 were chosen to represent fp constants and the float type, respectively.

Third, the float type was included as an allowed conversion value in the VC converter. It should be noted that the float type introduced is currently translated to type real for z3 interpretation. This should of course be modified to take advantage of z3's formal fp verification, but the float type needs to first be more completely integrated into Boogie.

Section 6: Working Examples

The following section is a list of functioning examples of floating points as of the time of this writing. Each example consists of the code and expected result:

Example 1, error expected:

```
procedure F() returns () {
        var x : float;
        var y : float;
        assert x == y;
}
```

Example 2, error expected:

```
procedure F() returns () {
        var x : float;
        var y : float;
        y := x - x;
        assert y == x;
}
```

Example 3, pass expected:

```
procedure F() returns () {
        var x : float;
        x := fp (0 0);
        assert x == fp (0 0 23 8);
}
```

Example 4, error expected:

```
procedure F() returns () {
        var x : float;
        x := fp (1.5);
        assert x == fp (1 0 23 8);
}
```

Section 7: Future Work

The most pressing issue is implementing remaining expected float functionality into Boogie. After float types work as expected within Boogie, it will be necessary to implement translation from Boogie floating types into z3 such that z3's formal verifier can actually be used for floating point values.

After float types are fully implemented into Boogie, the future of this project will be implementing the floating point type in SMACK, a verification language which uses Boogie.

References

What Every Computer Scientist Should Know About Floating-Point Arithmetic by Joe Darcy, https://oracleus.activeevents.com/2014/connect/sessionDetail.ww?SESSION_ID=2931&eventRef=javaone

Wikipedia article on floating-point arithmetic, http://en.wikipedia.org/wiki/Floating_point

Deciding floating-point logic with abstract conflict driven clause learning by Martin Brain; Vijay D'Silva; Alberto Griggio; Leopold Haller; Daniel Kroening, https://es-static.fbk.eu/people/griggio/papers/fmsd13.pdf

This is Boogie 2 by K. Rustan; M. Leino, http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf

Current github repository for project, https://github.com/Checkmate50/boogie