ITF1788: An Interval Testframework for IEEE 1788

Maximilian Kiesner, Marco Nehmeier, and Jürgen Wolff von Gudenberg

Institute of Computer Science, University of Würzburg
Am Hubland, D 97074 Würzburg, Germany
maximilian.kiesner@stud-mail.uni-wuerzburg.de
nehmeier@informatik.uni-wuerzburg.de
wolff@informatik.uni-wuerzburg.de

June 17, 2015

Abstract. Libraries implementing the interval arithmetic standard IEEE 1788 have to be tested for errors. Although the standard is language agnostic, the different implementations are not. Therefore the same tests must be implemented in many different languages. Hence it is useful to only have to formulate the tests once – in a domain-specific language – and to subsequently generate the language-specific test code automatically. To save time and encourage test-driven development, we present a language for testing interval arithmetic libraries and a corresponding source code generator which produces language-specific unit tests.

Keywords: domain-specific language, IEEE1788, interval arithmetic, ITF1788, source code generation, test-driven development, unit test

1 Introduction

Floating point numbers are widely used in computer science to approximate real numbers. Due to their limited precision, complex calculations may however lead to poor results. Interval arithmetic tackles this problem by representing a real value x as a tuple of floating point numbers $[a\,,\,b]$, called an *interval*, where $a \leq x \leq b$. Real-valued functions are adapted accordingly, requiring an interval-valued input and evaluating to an interval-valued output. The *Fundamental Theorem of Interval Arithmetic* then guarantees that the result of evaluating this function produces an interval which encloses the real-valued result of the original function.
The IEEE Interval Standard Working Group 1788 [7] is working on a comprehensive standard for interval arithmetic, which includes methods and algorithms, whose validity can be proven by mathematical methods. In practice the different language-specific implementations are prone to a variety of errors. These may occur in the implementation's source code, the compiler or the CPU. It is thus reasonable to spotcheck an implementation with unit tests.

The unit tests of the implementations differ in several aspects:

- The programming language they are written in
- The testing framework
- Idiosyncrasies, such as custom function names

However, a set of inputs has to fulfill the same requirements for the outputs among standard-compliant implementations. This leads to a recognizable pattern in common testing facilities which have a similar syntax. To avoid the necessity of writing a test suite for every implementation, we have developed a domain-specific language tailored to describing unit tests for interval arithmetic, called the Interval Test Language (ITL). Unit tests written in ITL can be passed to the Interval Test Framework for IEEE 1788 (ITF1788), which then generates unit tests for various target languages. In order to do this, the ITF1788 must have knowledge of the syntax of the language, the testing framework and the interval library. Figure 1 outlines how the ITF1788 works.

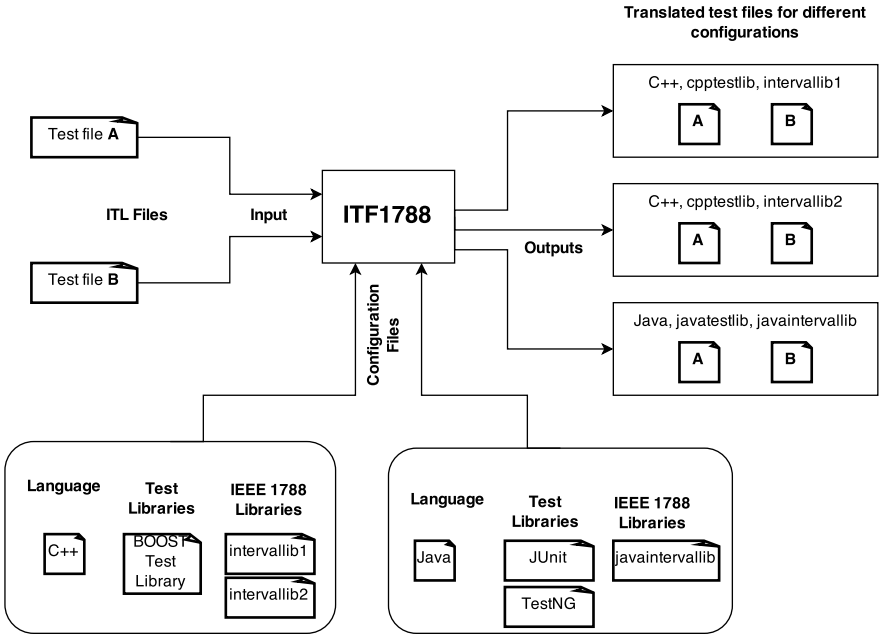

Fig. 1: Translation of ITL test files A and B for different implementations

This report aims to familiarize the reader with the ITL and the necessary configuration of ITF1788 to support a custom implementation.

2 The Domain-Specific Language ITL

Since our use case only has limited scope - i.e. ensuring that functions defined on intervals are implemented correctly - using a general-purpose language for writing the unit tests would likely result in code bloat and a non-intuitive syntax, which is contrary to what we wanted to achieve in the first place. The favorable approach is to use a syntax which:

- supports all of the elements necessary to test an implementation
- uses a domain-specific notation
- is easy to read and write

ITL satisfies these requirements. A minimal example of a test file is shown in Listing 1.

```
/*
 Testing the addition function
 */
testcase addition.test {
    add [-1.0, 1.0] [empty] = [empty];
    add [1.0, 2.0] [3.0, infinity] = [4.0, infinity];
    add [1.0, infinity] [-infinity, 4.0] = [entire];

    // using hexadecimal notation
    add [0X1.FFFFFFFFFFFFP+0] [0X1.999999999999AP-4] =
        [0X1.0CCCCCCCCCCC4P+1, 0X1.0CCCCCCCCCCC5P+1];

    # etc.
}

/*
 Testing the division function
 */
testcase division.test {
    div [empty] [empty] = [empty];
    div [-30.0, 15.0] [entire] = [entire];

    # etc.
}
```

Listing 1: Simple example of a test file

The following section provides a detailed description of the grammar of the ITL.

2.1 Grammar

In the following section terminal symbols are enclosed in single quotation marks and nonterminal symbols in angle brackets.

Both integral and floating point numbers are required for interval arithmetic. They may be represented as decimal or hexadecimal literals and can optionally be suffixed to set a specific data type. The number format is based on the well-known C99 standard [8].

Basic rules The following rules apply to the definition of integral as well as fractional numbers.

⟨*Digit*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

⟨*NonZeroDigit*⟩ ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

⟨*HexDigit*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'

⟨*HexPrefix*⟩ ::= '0x' | '0X'

⟨*Sign*⟩ ::= '+' | '-'

Integral Numbers

⟨*UnsignedSuffix*⟩ ::= 'U'|'u'

⟨*LongSuffix*⟩ ::= 'L'|'l'

⟨*LongLongSuffix*⟩ ::= 'LL'|'ll'

⟨*DecConstant*⟩ ::= ⟨*NonZeroDigit*⟩ ⟨*Digit*⟩*

⟨*HexConstant*⟩ ::= ⟨*HexPrefix*⟩ ⟨*HexDigit*⟩+

⟨*IntegerSuffix*⟩ ::= ⟨*UnsignedSuffix*⟩ ⟨*LongSuffix*⟩?
| ⟨*UnsignedSuffix*⟩ ⟨*LongLongSuffix*⟩
| ⟨*LongSuffix*⟩ ⟨*UnsignedSuffix*⟩?
| ⟨*LongLongSuffix*⟩ ⟨*UnsignedSuffix*⟩?

⟨*IntegerLiteral*⟩ ::= ⟨*Sign*⟩? ('0' | ⟨*DecimalConstant*⟩ | ⟨*HexConstant*⟩) ⟨*IntegerSuffix*⟩?

Floating Point Numbers

⟨*FloatingSuffix*⟩ ::= 'F'|'L'|'f'|'l'

⟨*HexSeq*⟩ ::= ⟨*HexDigit*⟩+

⟨*DigitSeq*⟩ ::= ⟨*Digit*⟩+

⟨*HexFracConst*⟩ ::= ⟨*HexSeq*⟩ '.' ⟨*HexSeq*⟩
| '.' ⟨*HexSeq*⟩
| ⟨*HexSeq*⟩ '.'

⟨*DecFracConst*⟩ ::= ⟨*DigitSeq*⟩ '.' ⟨*DigitSeq*⟩
| '.' ⟨*DigitSeq*⟩
| ⟨*DigitSeq*⟩ '.'

⟨*HexExpPrefix*⟩ ::= 'p' | 'P'

⟨*DecExpPrefix*⟩ ::= 'e' | 'E'

⟨*HexExpPart*⟩ ::= ⟨*HexExpPrefix*⟩ ⟨*Sign*⟩? ⟨*DigitSeq*⟩

⟨*DecExpPart*⟩ ::= ⟨*DecExpPrefix*⟩ ⟨*Sign*⟩? ⟨*DigitSeq*⟩

⟨*DecFloatConst*⟩ ::= ⟨*DecFracConst*⟩ ⟨*DecExpPart*⟩? ⟨*FloatingSuffix*⟩?
| ⟨*DigitSeq*⟩ ⟨*DecExpPart*⟩ ⟨*FloatingSuffix*⟩?

⟨*HexFloatConst*⟩ ::= ⟨*HexPrefix*⟩ (⟨*HexFracConst*⟩|⟨*HexSeq*⟩) ⟨*HexExpPart*⟩ ⟨*FloatingSuffix*⟩?

⟨*FloatingLiteral*⟩ ::= ⟨*Sign*⟩? ⟨*DecFloatConst*⟩
| ⟨*Sign*⟩? ⟨*HexFloatConst*⟩

Miscellaneous Rules

⟨*InfinityLiteral*⟩ ::= ⟨*Sign*⟩? 'infinity' ⟨*FloatingSuffix*⟩?

⟨*NaN*⟩ ::= 'NaN' ⟨*FloatingSuffix*⟩?

⟨*BooleanLiteral*⟩ ::= 'true' | 'false'

Intervals The program supports the inf-sup interval notation as well as constants for the empty, entire and NaI types.

⟨*IntervalBoundary*⟩ ::= ⟨*FloatingLiteral*⟩ | ⟨*InfinityLiteral*⟩

⟨*InfSupInterval*⟩ ::= '[' ⟨*IntervalBoundary*⟩ ',' ⟨*IntervalBoundary*⟩ ']'

⟨*EmptyInterval*⟩ ::= '[' 'empty' ⟨*FloatingSuffix*⟩? ']'

⟨*EntireInterval*⟩ ::= '[' 'entire' ⟨*FloatingSuffix*⟩? ']'

⟨*NaI*⟩ ::= '[' 'nai' ⟨*FloatingSuffix*⟩? ']'

⟨*DecorationLiteral*⟩ ::= 'trv' | 'def' | 'dac' | 'com' | 'ill'

⟨*BareInterval*⟩ ::= ⟨*InfSupInterval*⟩ | ⟨*EmptyInterval*⟩ | ⟨*EntireInterval*⟩

⟨*IntervalLiteral*⟩ ::= ⟨*NaI*⟩ | ⟨*BareInterval*⟩ | ⟨*BareInterval*⟩ '_' ⟨*DecorationLiteral*⟩

Arrays ITL supports array literals in a similar fashion to common programming languages.

⟨*ArrayElementLiteral*⟩ ::= ⟨*FloatingLiteral*⟩ | ⟨*InfinityLiteral*⟩ | ⟨*NaN*⟩

⟨*ArrayLiteral*⟩ ::= '{' '}' | '{' ⟨*ArrayElementLiteral*⟩ (',' ⟨*ArrayElementLiteral*⟩)* '}'

Identifiers Bare identifiers refer to operations, qualified identifiers to testcases.

⟨*alpha*⟩ ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'

⟨*identChar*⟩ ::= ⟨*alpha*⟩ | ⟨*digit*⟩ | '_'

⟨*ident*⟩ ::= ⟨*alpha*⟩ ⟨*identChar*⟩*

⟨*qualIdent*⟩ ::= ⟨*ident*⟩ ('.' ⟨*ident*⟩)*

Strings A string is a sequence of arbitrary characters (including line breaks) enclosed in double quotes.

⟨*StringLiteral*⟩ ::= ' " ' .* ' " '

Overlap Literals An overlap literal describes the realtionship between two intervals.

⟨*OverlapLiteral*⟩ ::= 'bothEmpty' | 'firstEmpty' |
'secondEmpty' | 'before' | 'meets' |
'overlaps' | 'starts' | 'containedBy' |
'finishes' | 'equals' | 'finishedBy' |
'contains' | 'startedBy' | 'overlappedBy' |
'metBy' | 'after'

Comments There are two distinct kinds of comments.

- Comments which are not translated and thus have no equivalent in the target source code. Single-line comments are prefixed by a `#` character, whereas block comments start with `#*` and end with `*#`.
- Comments which will be translated into language-specific comments. Line comments are initiated by `//`. Block comments start with `/*` and end with `*/`.

Test Files The basic elements of a test file, referred to as `Test` here, are statements which check if a function yields the expected result for the specified input values.

⟨*Literal*⟩ ::= ⟨*IntegerLiteral*⟩ | ⟨*FloatingLiteral*⟩ | ⟨*InfinityLiteral*⟩ | ⟨*IntervalLiteral*⟩ | ⟨*StringLiteral*⟩ | ⟨*OverlapLiteral*⟩ | ⟨*NaN*⟩ | ⟨*BooleanLiteral*⟩ | ⟨*ArrayLiteral*⟩ | ⟨*DecorationLiteral*⟩

⟨*OpName*⟩ ::= ⟨*ident*⟩

⟨*InputLit*⟩ ::= ⟨*Literal*⟩

⟨*TightestOutLit*⟩ ::= ⟨*Literal*⟩

⟨*AccurateOutLit*⟩ ::= ⟨*Literal*⟩

⟨*Outputs*⟩ ::= '=' ⟨*TightestOutLit*⟩+
| '<=' ⟨*AccurateOutLit*⟩+
| '=' ⟨*TightestOutLit*⟩+ '<=' ⟨*AccurateOutLit*⟩+

⟨*Test*⟩ ::= ⟨*OpName*⟩ ⟨*InputLit*⟩* ⟨*Outputs*⟩ ';'

Testcases are named blocks which may contain an arbitrary number of tests. They bear a name in the form of a qualified identifier.

⟨*TestCase*⟩ ::= 'testcase' ⟨*qualident*⟩ '{' ⟨*Test*⟩+ '}'

Finally, a test file may contain an arbitrary number of testcases.

⟨*TestFile*⟩ ::= ⟨*TestCase*⟩+

3 Setting Up ITF1788 in Linux

3.1 Installation

The ITF1788 is open-source software licensed under the Apache License [1]. You can obtain the source code by cloning our GitHub repository.

```
1 ~ $ git clone https://github.com/nehmeier/ITF1788
```

Listing 2: Cloning the GitHub repository

In order to run the program you need to have Python 3 [4] and pip (for Python 3) [2] installed. In a terminal, navigate to the application's root folder and type

```
1 ~/ITF1788 $ sudo pip3 install -r requirements.txt
```

Listing 3: Installing the required Python packages

This will install the Python packages PLY [3], which is used for the parser, and PyYAML [5], which is used for configuring the generator.
Note: *pip may have another name on your system. Change 'pip3' accordingly.*

3.2 Execution

The application's root directory contains a file `defaultGenerator.sh`. Running this will generate unit tests for all ITL files in the `itl/` folder and store them in the `output/` folder.
If you want to use custom options, e.g. a different output folder or generate output for a subset of ITL files only, you must run the `main` module in the `src/` subfolder with the desired options.

```
1 ~/ITF1788/src $ python3 main.py <OPTIONS>
```

Listing 4: Using custom console line options

For example we want verbose output, the directory of our ITL source file to be `/home/itf1788user/sources` and the generated files to be emitted to `/home/itf1788user/output`, the program would be called as follows:

```
1 ~/ITF1788/src $ python3 main.py -v -s "/home/itf1788user/sources"
      ↪ -o "/home/itf1788user/output"
```

Listing 5: Using custom console line options

For more information pertaining to the parameters and their syntax, call `src/main.py` with the `--help` option.

4 Configuration of the Test Generator

4.1 Folder Structure

To generate tests for an arbitrary target language, ITF1788 must obviously have knowledge of that language's basic keywords and syntactic elements. We differentiate between three distinct kinds of necessary information:

- The "core" of the language - keywords, use of brackets, implementation of primitive data types etc.
- The testing framework - structure of test files, assert statements etc.
- The IEEE 1788 implementation - intervals, custom function names etc.

We decided to take this approach as it offers great flexibility. Consider a language, a set of testing frameworks and a set of IEEE 1788 implementations. Instead of having to write a configuration file for every combination, we define the properties of the language, the testing frameworks and the interval standard implementations separately and let ITF1788 combine them as desired.
In the following example, we want to configure the test generator such that it supports a language L together with a test library T and an IEEE 1788 implementation I.

The required configuration files are stored in the `plugins/` folder. There we must first create a directory named "L". It must contain:

- a folder named `arith`
- a folder named `test`
- a file called `callbacks.py`, which is explained in subsection 4.2
- an empty file called `__init__.py`
- a file called `lang.yaml`, which contains the properties of the language L

The `arith` folder contains one subfolder for each of L's IEEE 1788 implementations that are supposed to be tested. Likewise, the `test` folder contains one subfolder for each of L's testing frameworks to be used. Finally, the properties of the libraries are defined in their respective subfolders, in the `arith.yaml` and `test.yaml` files. The final directory structure with additional testing frameworks and interval libraries may look as follows:

```
plugins
├── L
│   ├── arith
│   │   ├── I
│   │   │   └── arith.yaml
│   │   ├── I2
│   │   │   └── arith.yaml
│   │   └── I3
│   │       └── arith.yaml
│   ├── test
│   │   ├── T
│   │   │   └── test.yaml
│   │   └── T2
│   │       └── test.yaml
│   ├── callbacks.py
│   ├── __init__.py
│   └── lang.yaml
└── L2
    └── ...
```

Note: *A template for configuring a language can be found in the aforementioned GitHub repository in the* `templates/` *folder.*

4.2 Callbacks

By default, ITF1788 translates integer, floating point number, string and qualified identifier literals as they are provided in the ITL file, i.e. in a notation very similar to the C99 [8] syntax. If your target language requires a different representation, you can use Python functions to manipulate the output as needed. The functions are stored in the `callbacks.yaml` file. The following callbacks are supported:

- `cb_fpNum` - for floating point literals
- `cb_inp_var_name` - variable names for array inputs
- `cb_int` - for integer literals
- `cb_outp_var_name` - variable names for array outputs
- `cb_qualident` - for qualified identifiers
- `cb_string` - for strings

Assuming type suffixes for floating point numbers are not supported in your target language, you could include the following function in the callbacks.yaml file:

```
def cb_fpNum(val):
    # val is the representation in the ITL file -- check
        ↪ if there is a suffix
    if val[-1] in 'FLfl':
```

```
        # delete suffix
        val = val[:-1]
    return val
```

Listing 6: Callback function for floating point numbers

Note that if no callback method exists for a literal type, the default translation is used.

4.3 Format of the YAML files

The majority of the language specific configuration resides in YAML [6] files. YAML is a simple markup language which lets us store key-value pairs as text files, without having to care much about escaping characters. You can specify either single or multi-line values. The `#` character marks the beginning of a comment that extends to the end of the line.

```
# single line form
key1: value1

# multi line form
# The indentation is mandatory; an unindented line ends
   ↪ the multi line value
key2: |
        value2 line 1
        value2 line 2
        ...
```

Listing 7: Introductory YAML example

The user should be able to figure out most key-value pairs easily with the help of the template files, and if necessary an existing configuration in the `plugins/` folder. The following subsection covers function definitions as they appear in the `arith.yaml` files.

4.4 IEEE 1788 Functions in ITF1788

ITF1788 does not require a specific set of operations[1] that have to be configured in the `arith.yaml` file. This is helpful for two reasons:

- Some functions in IEEE 1788 are optional.
- No adaption of the source code is necessary if a new function is required.

Because of this design decision, functions may be refered to with arbitrary names. It is however strongly recommended to use the function names of the IEEE 1788 standard. The required operations of the standard are also included in the template `arith.yaml` file.

[1] Except for the `subset` and the `decorationPart` operation

ITF1788 omits the generation of assert statements if no matching operation is configured. If you run ITF1788 with the *verbose* option, warnings will be printed for each test in an ITL file for which no matching operation was found.

Functions may be distinguished by the types of both their input and output values. A function for adding two double-valued intervals which results in one double-valued interval is configured as follows:

```
op_add<<interval<double>>><interval<double>,interval<
    ↪ double>>: add(${ARG1}, ${ARG2})
```

Listing 8: Single line addition function with input and output types

This demonstrates several things:

- The key of an operation starts with 'op_' followed by the name of the operation
- The output types follow, as a comma separated list enclosed in doubled angle brackets
- Similarly, the input types are expressed as a comma separated list contained in single angle brackets
- The value specifies the function call with special placeholders representing its arguments, denoted by ${ARG1}, ${ARG2} etc.

To avoid the tedious task of having to retype an otherwise identical translation for several different data types, wildcard characters may be used to match a string of arbitrary length. Out of all matching operations, ITF1788 then chooses the one with the longest key (and thus the most specific one).

```
# exact match
op_add<<interval<double>>><interval<double>,interval<
    ↪ double>>: ...

# matches the add operation with an input of two double-
    ↪ valued intervals and arbitrary output types
op_add<<*>><interval<double>,interval<double>>: ...

# catches all other addition tests
op_add*: ...
```

Listing 9: Defining functions with wildcard characters

Also *one* assert statement in the ITL file may produce *several* assert statements in the generated test file. Say, for every addition test, you want to test the `add` method as well as the overloaded `+` operator of your implementation. You can then use the following configuration:

```
op_add*: |
        add(${ARG1}, ${ARG2})
        ${ARG1} + ${ARG2}
```

Listing 10: Multi line addition function

Some functions like `divToPair` produce several outputs. In comparison to listing 10, where the add method and the plus operator check for equality with the same (and only) output, we now need to be able to switch to the next output. For that, we insert a line containing '*** next output' at the desired position. Thus it is possible to write custom code for every output. Consider the following example:

```
op_divToPair*: |
    divToPair($ARG1, $ARG2).first
    anotherDivToPairImplementation($ARG1, $ARG2).first
    *** next output
    divTopair($ARG1, $ARG2).second
    anotherDivToPairImplementation($ARG1, $ARG2).second
```

Listing 11: Configuration of a function with multiple outputs

Lastly, a test may accept tightest or accurate literals or both at the same time. The generated source code differs as follows:

Tightest output literals The translation of the operation and the output literal are checked for equality with the `assert_equals` key-value pair of the testing framework. If the output is a decorated interval, the decorations are also checked for equality.

Accurate output literals The evaluation of the operation must be a subset of the output literal. Therefore, we use the `subset` operation and the `assert_true` key-value pair of the testing framework. If the output is a decorated interval, the decorations are checked for equality.

Tightest and accurate output literals
In this case the number of accurate and tightest output literals must be the same. ITF1788 produces the following tests for every pair of tightest and accurate literals:

- the result of the operation is equal to the tightest output. We use the `assert_equals_warning` key-value pair of the testing framework. If this fails, the target language does not abort execution but merely generates a warning.
- the tightest output is included in the result of the operation, tested with the `subset` operation.
- the result of the operation is included in the accurate output, tested with the `subset` operation.

If the output literals are decorated intervals, the decoration of the evaluated operation must be less-than-or-equal to the decoration of the tightest output and greater-than-or-equal to the decoration of the accurate output.

5 Conclusion

In this report we gave an introduction to the unit test generator ITF1788. By using the domain specific language ITL, we can now write tests for IEEE 1788 implementations in a simple and familiar notation and automatically generate unit tests for them. In comparison to writing tests for every implementation – all of which would basically do the same thing – we can use every test file to test any implementation. Besides increased productivity, it decreases the probability of errors in the unit tests, because they must pass on all of the various implementations.

References

1. Apache License 2.0, http://www.apache.org/licenses/LICENSE-2.0
2. pip, https://pypi.python.org/pypi/pip
3. PLY (Python Lex-Yacc), http://www.dabeaz.com/ply/ply.html
4. Python 3, https://www.python.org/download/releases/3.0/
5. PyYAML, http://pyyaml.org/
6. YAML Specification, http://yaml.org/spec/
7. IEEE P1788: Draft Standard For Interval Arithmetic. Tech. rep., http://grouper.ieee.org/groups/1788/
8. ISO: ISO C Standard 1999. Tech. rep. (1999), http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf, ISO/IEC 9899:1999 draft