FINAL REPORT

Correctly Rounded Floating-point Binary-to-Decimal and Decimal-to-Binary Conversion Routines for Standard ML

Rochester Institute of Technology
Department of Computer Science

Contents

Chapter 1

Introduction

1.1 The Problem

Dealing with real numbers is a vital part of modern programming in many scientific fields in which programming is used. However, the format in which humans and the computer understand real numbers are different. The most basic format of a real number that a human being can understand is a series of decimal digits (from 0 through 9) and a decimal point, such as 34.25 or 0.1. The computer, however, stores real numbers in a very different way, using binary digits (0 or 1) rather than decimal digits and performing arithmetic on the binary representation of the real number, then converting to a decimal representation that the user (a human) can read. In order to convert between the computer's representation (in binary) and the user's representation (in decimal), the computer must change the base with which the number is represented.

A simple algorithm exists that can convert easily between an integer in one base and an integer in another: for example, if we want to convert the decimal number 56 from base 10 to base 2, we can get the digits by repeated integer division by 2, adding the remainder onto the left side of an initially empty list as we go. When we hit 0, we've finished building the result.

Step	Number	Result
Start	56	[]
1	$56/2 = 28$	[$56\%2 = 0$]
2	$28/2 = 14$	[$28\%2 = 0$, 0]
3	$14/2 = 7$	[$14\%2 = 0$, 0, 0]
4	$7/2 = 3$	[$7\%2 = 1$, 0, 0, 0]
5	$3/2 = 1$	[$3\%2 = 1$, 1, 0, 0, 0]
6	$1/2 = 0$	[$1\%2 = 1$, 1, 1, 0, 0, 0]
End	0	[1,1,1,0,0,0]

Checking our work, we see that in fact it is true that $1*2^5+1*2^4+1*2^3 = 32+16+8 = 56$ is indeed the correct answer. We label this as a base-2 representation with an appropriate subscript:

$$111000_2$$

However, this is not as simple when converting between the decimal and binary bases for real numbers. For example, say a human asks a computer to convert the number 0.3125 from a decimal to a binary representation. Could we modify our above algorithm to work? We could modify it to multiply the number by a power of 2 so that it is an integer and then perform the above algorithm, moving the decimal point over at the end to compensate for our initial multiplication. We find that $0.3125*2^4 = 5$, and then we run our algorithm as above,

Step	Number	Result
Start	5	[]
1	$5/2 = 2$	[$5\%2 = 1$]
2	$2/2 = 1$	[$2\%2 = 0$, 1]
3	$1/2 = 0$	[$1\%2 = 1$, 0, 1]
End	0	[1,0,1]

Now, since what we have discovered using our algorithm is actually $101_2 = (0.3125*2^4)_{10}$, we shift the decimal point to the left 4 places to remove the factor of 2^4 to arrive at

$$0.0101_2 = 0.3125_{10}$$

However, this case is rather innocuous - there are many others where not only will this simple algorithm not work out, but where there is no finite representation in base 2 of a

simple number in base 10. For example, say the user now asks the computer to 0.1 from a decimal to a binary representation. The computer searches in vain for a power of 2 to multiply 0.1 by in order to achieve an integer, but will never find one - in fact, there is no such integer. In fact, the simple decimal case we are attempting to convert to binary has no finite binary representation! The exact conversion of 0.1_{10} into a base-2 representation yields the infinite string

$$(0.000110011001100...)_2 = 0.000\overline{1100}_2$$

In general, since the computer has to fit the result of the conversion in a finite space, it cannot perform the exact conversion in this case; rather, it should settle for a number that is as close as it can get to the actual answer in the space it has available. However, clearly we need more sophisticated algorithms to solve this problem and choose a close representation where no exact one exists.

For the conversion from binary to decimal, it is always possible to exactly represent the binary number in base 10, but this is often not what we want - for example, the binary format stored by the computer is capable of storing a binary representation of the value 2^{-1074} in a 64-bit word; however, the exact decimal representation of this number, `4.940656...E-324` has over *700 decimal digits*! It isn't necessary for all of them to be output, just enough so that the computer will choose the original binary representation as the closest possible to the input when converting back to decimal. In this case, that is simply `5E-324`.

1.2 THIS PROJECT

Some have argued that conversions between binary formats stored by computers and decimal real number formats that are easily readable and writable by human programmers and users are sufficiently similar to the point where they can both be treated similarly [Jaf15]. After all, both conversions are, at their core, a base conversion from some source base b to a destination base B. However, this treatment does not seem to adequately describe the problem (as the storage strategies for the binary and decimal representations used in modern computers differ significantly) nor does it provide for a particularly efficient solution (certain optimizations can be performed to increase efficiency of these operations). Namely, the binary-to-decimal has many sub-problems, such as the fixed-length problem, in which the decimal representation we seek is rounded to a certain number of significant digits, or the exact-length problem, in which we seek the exact decimal representation of the binary real input. The only sub-problem addressed by this project is the shortest length output problem,

which involves returning the shortest possible decimal representation that preserves the original binary representation information (i.e., so we can retrieve the original number with a correctly rounded conversion back).

The goal of the software developed in this project is to offer a native implementation of these conversions in Standard ML, in the hopes of allowing the implementations to be integrated into the MLton compiler, replacing an external dependency on the `gtdoa` C library [Gay] currently used for the conversions, which is both a nagging dependency and generalizes the conversion process more than is required by the compiler.

The algorithms implemented for this case were the Ryū algorithm for the binary-to-decimal conversion [Ada18] and an algorithm published by Aubrey Jaffer [Jaf15] for the decimal-to-binary conversion. A test structure was also created to test for correctness and to perform time comparisons as necessary to describe the performance of the methods. The implementations successfully rounded correctly in tens of thousands of tested cases, as well as outperformed the current implementation in some cases; however, there is still work to be done before the project is ready to be integrated into MLton and replace the `gtdoa` library.

1.3 Binary and Decimal Formats

The decimal format that a human can read and write is extremely liberal - it can be any length, and many equivalent representations of the same number exists. For example, the number `0.002` is the same as the number `2E-3`, or the number `0.2E-2`. A human could also pass in a decimal representation 2.145295648... containing 200 digits and expect the computer to properly handle the input.

The binary representation stored by the computer, on the other hand, is of a fixed size and format that is standardized; in particular, most computers operate on and store IEEE-754 standard real floating point binary representations of real numbers, which is the format that we deal with in this project. An IEEE-754 binary representation is a word containing a certain number of bits, sections of which serve different functions. For most applications, and for those covered in this project, this number of bits is either 32 (called “single precision") or 64 (called “double-precision"). The three main sections of the real representation are the sign s, the mantissa m_s, and the exponent e_s, that together represent the floating point number.

1.3.1 Sign

The sign is stored in a single bit, with 1 indicating that the number is negative and 0 that it is positive. Every numerical value represented by the exponent and mantissa have a positive and negative version, *including 0*. There are several important reasons for including a signed 0 rather than keeping only a single unsigned representation [Gol91], but those reasons are beyond the scope of this paper.

1.3.2 Mantissa

The mantissa is the representation of the coefficient multiplied by a power of 2 represented by the exponent to arrive at the real value. It is stored in 23 bits in the single-precision case and 52 in the double-precision case. For the majority of cases, the mantissa also has a single “hidden bit" that is not stored. This bit is to the left of the other bits, and is the sole bit to the left of the decimal point. The value of this digit is implied by the exponent; in most cases, the bit is implied to be 1, but if the exponent is 0, then the bit is implied to be 0. Therefore, for example, if the mantissa was stored in 4 bits rather than 23 or 52, a stored mantissa of 0110 would represent the value $m_r = 1.0110_2 = 1.375_{10}$ if the exponent were non-zero and the value $m_r = 0.0110_2 = 0.375_{10}$ if it were 0. This paper will often refer to the *stored mantissa* and the *represented mantissa* to differentiate between the value that is stored and the value that is represented or the number of bits in each.

1.3.3 Exponent

The exponent is a representation of the power of 2 that scales the coefficient defined by the mantissa. It is stored in 8 bits in the single-precision case and 11 in the double-precision case.

Each precision of the standard also includes an exponent *bias* - an unsigned integer constant b used to convert stored unsigned integer exponents into represented signed exponents. For the single-precision case, the bias is 127, and for the double-precision case it is 1023. Similarly to the mantissa, this separates the *stored* exponent, an unsigned integer, from the *represented* exponent, which is signed.

The exponent also separates out two important special exponent cases:

1. Infinity and NaN - Values when the stored exponent is its maximum possible value (i.e. all ones) are non-finite cases. The sub-case where the stored mantissa is all zeroes represent $\pm\infty$ (based on the stored sign bit), and sub-cases where the mantissa is not all zeroes represent `NaN` (regardless of the sign).

2. Subnormal values - the case where the stored exponent is 0 implies that the hidden mantissa bit is 0 as discussed above. These cases are called “denormal" or “subnormal", and in this case the represented exponent value is taken to be the same as the represented exponent value for the case where the exponent is 1. [Gol91]

Thus the actual represented value (in cases that are finite) is given as

$$e_r = \begin{cases} e_s - b, & e_s > 0 \\ e_s - b + 1, & e_s = 0 \end{cases}$$

1.3.4 Putting It All Together

Combining all of the information about the representations of different parts of the stored IEEE-754 binary format, the real number represented by the stored IEEE-754 float $f_s = (s, e_s, m_s)$ is

$$f_r = \begin{cases} \text{NaN}, & e_s = 11...11, m_s \neq 0 \\ (-1)^s * \infty, & e_s = 11...11, m_s = 0 \\ (-1)^s * 1.m_s * 2^{e_s - b}, & 0 < e_s < 11...11 \\ (-1)^s * 0.m_s * 2^{e_s - b + 1}, & e_s = 0 \end{cases}$$

Chapter 2

Algorithms

2.1 Preliminaries

This chapter discusses the algorithms used for testing and implementation, including the binary-to-decimal and decimal-to-binary conversion algorithms and important testing algorithms.

For the conversion algorithms, we note that dealing with the sign is trivial - it is impossible for a decimal representation of one sign to round to a binary representation of a different sign, since even if we have a case where a signed decimal representation rounds to zero, signed zero will assure that the sign remains the same. Therefore, often when discussing these algorithms, we will make the assumption that all values are positive, since that is how both algorithms treat them. The sign can be safely recorded before the conversion takes place and then put back in at the end.

This chapter will give special attention to the Jaffer (decimal-to-binary) algorithm, since it was modified due to issues with the algorithm in the original published paper.

2.1.1 Pseudocode

The algorithms are discussed where appropriate using a modified hybrid of SML-like and C-like pseudocode. `if` statements generally will be treated as expressions as in SML, but in general `let` blocks will be spilled out into lists of definitions with following return statements.

2.1.1.1 Integer Types

The pseudocode algorithms in this section make no distinction at all between integer types, leaving this detail up to implementation; instead, they make use of an infinite-precision

integer type, *Integer*, that is replaced in the actual implementation of these algorithms with types appropriate to the size needs of the variables.

2.1.2 `Real7`

To assist in illustrating certain algorithms discussed here, a “toy" real type similar to the IEEE-754 real types supported by these implementations will be used to display operation of the algorithms without using the large numbers that would be required with single- or double- precisions. The type is, as the name suggests, stored in a 7-bit unsigned integer type. The type will have 1 sign bit, 3 exponent bits and 3 mantissa bits, with the usual reserved exponents for non-finite and subnormal cases discussed in section 1.3. The bias b_7 is 3. We note that the exponent for a represented mantissa value interpreted as a `Real7` (i.e. an integer with the number of bits in the represented mantissa cast to a `Real7`) will have a (biased) exponent of $110_2 = 6$.

2.1.3 Predefined Parameters

Some algorithms use certain predefined parameters that we will define here:

- *rep_mantissa_length*, *stored_mantissa_length* - the number of bits in the represented and stored mantissa, respectively. As a reminder from section 1.3, the represented mantissa has the same number as the stored mantissa, plus one extra for the additional “hidden" bit.

2.2 Test Generation

The basic testing algorithms are designed to perform tests for both the binary and decimal-bound conversion algorithms, both for correctness and for time taken, for a given implementation of the conversion algorithms.

The algorithms can be used to generate “flash-card-like" tests that maintain both a binary and decimal value, and can theoretically be used to test both types of conversions implemented. However, recall that our algorithms solve only the shortest-length output problem in the binary-to-decimal case, as discussed in section 1.2). As a result, randomly generated *decimal* tests may not be capable of being converted back to the original decimal value if the decimal value chosen for the test is not the shortest possible information-preserving output. Therefore, while tests generated using randomly generated binary representations can be

used for both types of testing, we make the assumption that tests generated using random decimal representations can be used only to test decimal-to-binary conversions.

Besides the cases randomly generated, other contrived tests were done along the way to ensure that simple edge cases, like infinities, boundaries between representable IEEE-754 exponents, minimal or maximal subnormal cases, and maximal positive exponents, were rounding correctly. This was to ensure that the implementation is running properly in cases when work not required by a normal case is done (for example, NaN cases are handled separately from other cases, and cases that are far beyond the representable range and round to infinity in the decimal-to-binary case must be succesfully identified and removed rather than proceeding with the algorithm and returning a degenerate case with an exponent that is too high).

The test generation algorithm has two parts, which together allow for random data to be generated and conversions performed on that data by a given implementation to be checked.

2.2.1 Random Representation Generator

This algorithm uses a random number generator or source of some kind to generate both random binary and random decimal representations. The probability of choosing a given test from among the set of valid inputs in a certain reasonable range (roughly corresponding to the range of representable finite, non-zero real values) should be uniform up to certain considerations in the decimal case (since there are infinitely many decimal representations, truly uniform selection over this entire set is not mathematically possible - for example, there is a 0% chance of selecting a representation that is too long to be stored in the computer).

The binary algorithm is trivial - since all words of the size stored by the real type are valid real values, selecting random words of the correct size is all that is needed up to compressing all NaN values into a single value. Correct rounding implies that tests generated in this fashion are suitable for both conversion tests.

The decimal algorithm is more nuanced - again, as there are infinitely many decimal values, and many more different types of representations relative to the actual representable values, multiple types of test generation are appropriate. For example, there are obvious “edge cases" where there is not one choice for the nearest representable binary floating point number to a given decimal representation, but two. In this case, the algorithm employs a technique called “round to even", rounding to the real value with an even mantissa. Testing that this behavior is operating properly, and that the algorithm is operating properly on other cases nearby are also is important, as the case itself may involve different behavior than other

cases, and the nearby cases show that the transition from rounding from one representable number to rounding to the next is going smoothly.

For example, the following case is an edge case for our contrived `Real7` type: consider the problem of converting the decimal representation 5.25 into our type. The potential representations

$$(s, m_s, e_s) = (0, 010_2, 101_2) = 1.010_2 * 2^{101_2 - b_7} = 1.25 * 2^{5-3} = 1.25 * 2^2 = 5.0$$

and

$$(s', m'_s, e'_s) = (0, 011_2, 101_2) = 1.011_2 * 2^{101_2 - b_7} = 1.375 * 2^{5-3} = 1.375 * 2^2 = 5.5$$

are both 0.5 from the represented case, and no other representations are closer. Therefore the algorithm chooses the representation with a 0 in the leftmost significant bit and rounds to $(s, m_s, e_s) = 5.0$.

2.2.1.1 Types of Random Generation

The types of random representation generation implemented for this project were as follows. Random sign bits were generated for each test type.

1. **Fixed-length random generation:** randomly generated exponents and strings of decimal digits of a length passed in as a parameter are generated.

2. **Binary-Representation-based random generation:** Generating these tests involves carrying out a naïve version of a binary-to-decimal conversion that gets the entire representation rather than the shortest length output so that it can be tested by the decimal-to-binary converter. The test operates by randomly generating a binary representation, and then carrying out the naïve process to arrive at its perfect decimal representation.

```
algorithm COMPUTE-EXACT-REPRESENTATION (Integer mantissa, Integer exponent):
    1) // Find the binary exponent of the least significant bit of the mantissa
       Integer epsilon_exp = exponent − stored_mantissa_length
    2) Integer epsilon_decimal =
           if (epsilon_exponent < 0) then
               POWER (5, −epsilon_exponent)
```

else

POWER (2, *epsilon_exp*)

3) *Integer final_digits* = *epsilon_decimal* $*$ *mantissa*

4) *Integer final_decExp* =

if (*epsilon_exp* < 0) **then**

DECIMAL-LENGTH (*final_digits*) − **DECIMAL-LENGTH** (*epsilon_exp*)

+ **FLOOR** ($log_{10}2 * epsilon\_exp$) +1)

else

DECIMAL-LENGTH (final_digits)

5) **return** (final_digits, final_decExp)

This algorithm can be used on its own to generate exact decimal representations of binary reals; it can also easily be modified in order to allow for edge case generation - an edge case is directly between two normal cases, so we need to

(a) Choose a random mantissa and exponent

(b) Decrease the exponent by 1, and add a 1-bit to the right side of the mantissa

(c) Call **COMPUTE-EXACT-REPRESENTATION** on the modified inputs (this algorithm operates on mantissas of any length, so passing in a mantissa that is one more bit is fine).

2.2.2 Trusted Conversion Implementation

A trusted conversion implementation that generates the other side of a test (decimal *or* binary) when given one side as input. Upon querying this algorithm with a correctly formatted input, it computes the correctly-rounded result of the conversion. Given random data, this gives us access to an accurate conversion performed on that data to compare against for correctness.

As a small implementation note, ideally this is an algorithm that has already been well-tested and widely used, as implementing another algorithm as the trusted converter for this project would be an exercise in futility (since the problems handled in the project are non-trivial to the point where the complexity of this algorithm would require the introduction of yet another algorithm to check its correctness). Therefore, in the implementation of the test structures, the currently built-in gtdoa library was used as the trusted conversion algorithm, and therefore tests of the new implementation were done against it for correctness.

2.3 RYŪ (BINARY-TO-DECIMAL)

The Ryū algorithm, published in 2018 by Ulf Adams, is a decimal-to-binary conversion algorithm that uses heavy optimizations based on the fact that only the shortest information-preserving output is required. Because of this fact, a large amount of calculation can be performed beforehand and stored in a table that can be accessed at compile- and run-time. The paper published by Adams outlines the following basic steps of the binary conversion algorithm:

1. Remove special standalone cases (0, ∞ and `NaN` cases)

2. Unify the remaining representations into a format representing the mantissa as an integer by modifying the exponent, “shifting the decimal point" to the end of the mantissa.

   We will also find the *interval of information-preserving outputs*, which are those values that will round to the initial value put in when converted back. The interval consists of three points - the center point will be an integer v representing the actual value, and the points to the left (u) and the right (w) will be integers that represent the edge-cases between the value represented by the binary value passed in and the adjacent representable values below and above, respectively. Since the value passed in could be represented by a mantissa of $v = 1000..._2$, the next lowest point u could have a lower exponent by 1. In that case, we would need

   (a) One extra bit because u is all 1s and has the same bit length as v (namely the number of bits in the represented mantissa) but an exponent that is one lower

   (b) One extra bit because the edge case will have this many bits plus one (for example, in `Real7`, the edge case between the representable values $1.000_2 * 2^1$ and $1.111_2 * 2^0$ is $1.1111_2 * 2^0$, with 5 bits instead of the 4 in the full represented mantissa).

   We should, however, keep in mind that if this is a subnormal case, there is nothing done to

   This motivates the following computation for the interval of information-preserving outputs [Ada18]. Take $B = (s, m_s, e_s)$ as the binary IEEE-754 input, as well as e_r as the unbiased exponent and m_r as the represented mantissa. We define the adjusted exponent e_2 of all 3 of the endpoints as

$$e_2 = e_r - 2$$

and the actual endpoints as

$$u = 4m_r - \begin{cases} 1, & m_s = 0_2 \text{ and } e_s > 1_2 \\ 2, & \text{otherwise} \end{cases}$$

$$v = 4m_r$$

$$w = 4m_r + 2$$

3. Convert the representations in this interval to decimal. This is the heavily optimized step in the algorithm, since there are many possible values that we can convert to that will yield the correct result in the end, although some may take much more processing than others. Ryū's main contribution is that the algorithm pre-computes numbers that can be used to shorten the outputs of this algorithm. Since we need only the shortest possible decimal representation (and as we saw previously, the exact decimal representation could have hundreds of significant digits), we can in many cases remove most of the digits in the exact representation so that the process that finds the shortest possible algorithm takes far less time. Ryū optimizes this process even further by removing the need to compute the exact representation in the first place, rather storing quotients of large (arbitrary-precision) powers of 5 and 2 for each final base-10 exponent value in a lookup table. Although the powers of 5 and 2 are large, the quotients can be fit into fixed precision integer types when the algorithm is implemented. The algorithm operates by multiplying the appropriate value for the exponent input by each of the three interval points, u v and w, and then the result is scaled by some power of 2. The result is the digits of the exact representation divided by some large power of 10. Adams proves that the power chosen for each exponent is just large enough to ensure that the output list still has enough information to describe the binary input that we started with when converting back. The resulting interval points are called a, b and c, each corresponding to u, v and w, respectively.

   The exponent e_{10} is computed as follows:

$$e_{10} = \begin{cases} 0, & e_2 = e_2 \geq 0 \\ e_2, & e_2 < 0. \end{cases}$$

4. Find the shortest correctly-rounded decimal representation of the binary number. With most digits removed by the previous step, we pass the remaining interval into an

algorithm that removes more digits one-by-one until the center point of the interval, at this point represented by b, either has 1 digit left or is equal to one of a or c. Because we round using round-to-even, at this point b is either even (in which case a or c is an acceptable choice - they will round to the initial input) or else one more digit is needed to distinguish between b and the two endpoints (which will round away from the initial input).

2.4 JAFFER'S ALGORITHM (DECIMAL-TO-BINARY)

The decimal-to-binary case algorithm implemented is a modified version of an algorithm due to Aubrey Jaffer and published in 2015. The fact that the original algorithm was found not to round correctly in certain subnormal cases led to modifications to deal properly with these cases.

2.4.1 Primary Algorithm

The original algorithm takes in an integer representing the mantissa and an exponent (such that the actual real value represented by mantissa m and exponent e is $m * 10^e$) and computes the resulting binary real representation. This section will discuss the algorithm, then go into more depth with some issues that can be found in the original algorithm when dealing with subnormal cases, as well as a modified version of the algorithm adapted by this implementation to account for those issues.

Another notable change is what will be our choice for input: although all integers in this section are treated as infinite precision, Jaffer's algorithm as given in her paper takes in only Java `long` values for the mantissa digits. We avoid this restriction because it guarantees that some cases will not be able to be represented correctly; for example, as discussed in the 2.2 section, we must properly deal with edge cases. One such case is the double-precision edge case between the minimum positive subnormal double and zero, which has over 700 digits in its exact scientific notation representation. Round-to-even dictates that this edge case should round to zero; however, if even a single 1 is added to the end of the list of digits, then the resulting case should round up to the minimum positive subnormal double. This important level of detail is lost by confining the input to any fixed-precision type (assuming, of course, that there is no practical fixed-precision type capable of storing 700 decimal digits exists).

Operation

The main methodology of the algorithm is to manufacture a correctly rounded division (rounding to the nearest quotient, with round-to-even to break ties) that yields an integer fitting perfectly into a represented mantissa, which can then be cast to a real and then scaled to the correct exponent without any data loss. The algorithm takes in a parameter *mant*, an integer representing the base-10 mantissa digits, and *point*, an integer representing the base-10 exponent (so the value represented is $mant \times 10^{point}$).

Although the original algorithm scales values by powers of 10, the implemented version is an equivalent modification given by Jaffer that uses powers of 5 rather than powers of 10, and adds in a compensatory scale by the power of 2 factor difference between the two when scaling the final binary representation (which will be the absolute value of the *point* parameter). This reduces the size of intermediate integers by 29% [Jaf15]. While we will discuss the algorithms with the powers of 10 (as this will make the reasoning of the algorithm simpler to understand), the pseudocode given will use the powers of 5 optimization (since these were the algorithms implemented).

The algorithm is split into two cases based on whether or not the *point* parameter is negative or not - we will cover each of the cases in its own section because of the extent to which they have been treated differently in the implementation from the original algorithms.

2.4.1.1 Non-Negative Case

The non-negative case operates as follows:

1. Find the actual represented value by multiplying the *mant* by the power of 10 represented by *point*.

2. If the represented value has the number of bits of a binary represented mantissa already (or fewer), then we've found the answer already - cast it to a binary representation immediately and return (after possibly scaling the exponent if more bits had to be added to the left side to get the correct size for a represented mantissa)

   Otherwise, we need to shift down the value to the size of a mantissa. We perform a rounded quotient of the value from step 1) by a power of 2 spanning the difference between the bit size of the value and the represented mantissa length, and then scale the exponent by the number of bits we removed from the number.

Below is the power-of-5-optimized version of the non-negative algorithm:

algorithm **JAFFER-NON-NEGATIVE** (*Integer mant*, *Integer point*):

1) *// point is non-negative*
   Integer num = *mant* ∗ **POWER**(5,*point*)
2) *Integer bex* = **BIT-LENGTH** (num) − *rep_mantissa_length*
3) **return**
   if (*bex* <= 0) **then**
   // Real-ScaleB scales a real by a power of 2
   REAL-SCALEB (**REAL-VALUE** (*num*), *bex*)
   else
   let-expression:
   1) *Integer quo* = **ROUND-QUOTIENT** (*num*, **POWER** (2, *bex*))
   2) **return** **REAL-SCALEB** (**REAL-VALUE** (*quo*), *bex* + *point*)

We'll trace the algorithm for a `Real7` case: we aim to convert $1.1 * 10^1$ to `Real7`. We note that the *mant* parameter is 11, and the *point* parameter is 0.

1. *num* = 11 * 5^0 = 11 * 1 = 11
2. *bex* = **BIT-LENGTH**(11) - *rep_mantissa_length* = **BIT-LENGTH**(1011_2) - *rep_mantissa_length* = 4 - 4 = 0
3. *bex*≤ 0 ⇒ answer = **REAL-VALUE**(11) = ($s = 0, m_s = 011_2, e_s = 110_2$)

and we can easily check that this is indeed the `Real7` equivalent of 11.

2.4.1.2 Negative Case

The negative case is slightly different, but with the same ultimate goal. It operates as follows:

1. Find the power of 2 that *mant* must be multiplied by so that dividing the result by the power of 10 represented by *point* (to arrive at the closest possible representable real to the value we seek to convert, but scaled up by a power of 2) will result in an answer that is either *rep_mantissa_length* bits or *rep_mantissa_bits*+1 bits.
2. Scale the *mant* parameter by the power of 2 found, and perform the rounded-quotient by the power of 10 computed.
3. If the result is too large, then redo the calculation with the power of 10 multiplied by 2.
4. Scale the result as necessary as in the non-negative case.

2.4.1.3 Minor Modifications to the Negative Case

The negative algorithm given here is almost the same as the power-of-5-optimized one given by Jaffer; the only differences are the following, both related to the fact that we deal with mantissa integers of arbitrary bit length rather than the `long` integers of Jaffer's version.

Since we deal with values that could have a huge number of digits, the *bex* parameter may become positive when using powers of 5 rather than powers of 10, a phenomenon that has been seen in tests of the implementation, but was not planned for by Jaffer. Since this could cause us to lose data if we naïvely shift *mant* to the right, we avoid this by introducing another algorithm that takes in 2 parameters, the numerator and denominator of a to-be-performed rounded quotient, and left-shifts the denominator if the shift passed in is negative rather than right-shifting the numerator:

algorithm **QUOTIENT-SCALE-POW2** (*Integer numerator*, *Integer denominator*, *Integer bex*):

1) *Integer pow2* = **POWER** (2, |bex|)
2) **return**
   if (*bex* >= 0) **then**
      (*numerator* ∗ *pow2*, denominator)
   else
      (numerator, *denominator* ∗ *pow2*)

The other addition has to do with the rounded quotients themselves: we know that the standard integer quotient of two numbers n and d, of B_n and B_d bits, respectively, could have either $B_n - B_d$ or $B_n - B_d + 1$ bits; however, in a rounded quotient case, it is possible that the quotient rounds up to the nearest number, which could be the next highest power of 2, therefore having $B_n - B_d + 2$ bits. Jaffer's fix for this problem is to check all possible cases where this could occur in her algorithm to ensure that such a case never occurs, therefore showing that this concern does not apply, and thus that only a maximum of 2 rounded quotients are required for the computation, where the potential second is to correct for a result from the first that fails to fit into the correct number of bits [Jaf15]. Since we adopt her algorithm for multiple real precision types and also deal with *mant* values that could be theoretically any length, we do not have the luxury of testing all possible cases.

However, this turns out to not be a serious issue; even if there is such a case (one has never been observed), the round-to-even being 2 bits above the length of the represented mantissa means that the case rounded up to $2^{rep\_mantissa\_length+1}$, and therefore the actual

(rational) quotient result must have taken the form

$$2^{rep\_mantissa\_length+1} - x,\ x \in (0, 0.5],$$

and therefore by shifting the denominator left by 1 bit we cut the rational quotient result in half, yielding

$$2^{rep\_mantissa\_length} - \frac{x}{2},$$

which will also round up to a power that is too large to fit into the represented mantissa in this case. Therefore, we must shift the denominator left by 2 bits, and still only need perform one additional round quotient. This functionality is built into our modified algorithm below by replacing the original adjustment to *quo* and *bex*, a left shift of the *adjusted_scale* by 1, with a left shift by the *adjust* parameter instead, which could theoretically be 1 or 2.

algorithm **JAFFER-NEGATIVE** (*Integer mant*, *Integer point*):

1) *// point is negative*
   Integer scale = **POWER** (5, −*point*)
2) *Integer bex* = **BIT-LENGTH** (*mant*) − **BIT-LENGTH** (*scale*) − *rep_mantissa_length*
3) (*Integer num*, *Integer adjusted_scale*) = **QUOTIENT-SCALE-POW2** (*mant*, *scale*, −*bex*)
4) *Integer quo* = **ROUND-QUOTIENT** (*num*, *adjusted_scale*)
5) *Integer adjust* = **BIT-LENGTH** (*quo*) − *rep_mantissa_length*
6) (*Integer adjusted_quo*, *Integer adjusted_bex*) =
   if (*adjust* > 0) **then**
   (**ROUND-QUOTIENT** (num, adjusted_scale ∗ **POWER** (2, *adjust*)),
   bex + *adjust*)
   else
   (*num*, *scale*)
7) **return** **REAL-SCALEB** (**REAL-VALUE** (*adjusted_quo*), *adjusted_bex* + *point*)

We will trace the modified negative algorithm discussed above for a `Real7` case on the decimal representation $2.12 * 10^0$. We note that *mant* is 212 and *point* is −2.

1. *scale* = **POWER**(5,2) = 25
2. *bex* = **BIT-LENGTH**(412) − **BIT-LENGTH**(25) − 4 = 8 − 5 − 4 = −1
3. (*num*, *adjusted_scale*) = **QUOTIENT-SCALE-POW2**(212, 25, 1) = (414, 25)

4. *quo* = **ROUND-QUOTIENT**(414, 25) = 17
5. *adjust* = **BIT-LENGTH**(17) - 4 = 5 - 4 = 1
6. *adjust* $> 0 \Rightarrow$ (*adjusted_quo*, *adjusted_bex*) = ((**ROUND-QUOTIENT**(414, 50) = 8), $(-1 + 1 = 0)$
7. answer = **REAL-SCALEB**(**REAL-VALUE**(8), $0 + -2$) = **REAL-SCALEB**(($s = 0, m_s = 000_2, e_s = 110_2$), -2) = ($s = 0, m_s = 000_2, e_s = 100_2$)

This corresponds to the real value 2.0, and given that the next highest representable binary value corresponds to the real value 2.25, this is definitely the closest representation we could have chosen.

2.4.2 Subnormal Rounding Issues

The algorithm is based on the idea that the rounded quotient that yields the mantissa can be cast to a real directly and have the binary exponent modified directly, this is not true of subnormal values, which are not the same number of bits. While it may seem that we can just shift the mantissa to the right (or more appropriately perform a round quotient) to fix the number of bits if the scaling runs to an exponent below the maximum normal exponent, this turns out not to be true; there are cases where this strategy is not enough to fix the issue, and where the current **JAFFER-NEGATIVE** algorithm will round incorrectly.

We offer an example in `Real7`: consider the case $D = 0.08203125 = 0.0625 + 0.015625 + 0.00390625 = 2^{-4} + 2^{-6} + 2^{-8}$. This is close to the edge case $0.0625 + 0.015625 = 0.078125$, between 0.0625 and 0.09375. Therefore, the nearest binary `Real7` representation to E, which we will call B, is 0.09375, which has the binary format

$$(s, e_s, m_s) = (0, 000_2, 011_2),$$

and since the case is subnormal and the exponent is 0, the value represented by B is

$$0.11_2 * 2^{-2}.$$

Writing D in the same way, we have

$$D = 0.10101_2 * 2^{-2}.$$

The **JAFFER-NEGATIVE** algorithm will operate as follows on input D, and note that whether or not we shift left or use a (more accurate) round quotient at the end to scale the exponent beyond the minimum normal exponent, we will still round down incorrectly. We note that *mant* = 8203125 and *point* = −8.

1. *scale* = **POWER**(5,8) = 390625
2. *bex* = **BIT-LENGTH**(8203125) − **BIT-LENGTH**(390625) − 4 = 23 − 19 − 4 = 0
3. *quo* = **ROUND-QUOTIENT**(8203125, 390625) = 21
4. *adjust* = **BIT-LENGTH**(21) - 4 = 5 - 4 = 1
5. *adjust*> 0 ⇒ (*adjusted_quo*, *adjusted_bex*) = ((**ROUND-QUOTIENT**(8203125, 781250) = 10), (0 + 1 = 1))
6. answer = **REAL-SCALEB**(**REAL-VALUE**(10), 1 + (−8)) = **REAL-SCALEB**(($s = 0, m_s = 010_2, e_s = 110_2$), −7) = **REAL-SCALEB**(($s = 0, m_s = 010_2, e_s = 001_2$), −2) = ($s = 0, m_s =$ **ROUND-QUOTIENT**$(1010_2, 2^2), e_s = 000_2$)) = ($s = 0, m_s = 010_2, e_s = 000_2$)

Therefore we can see that by rounding to the nearest *normal* represented mantissa value in this case yielded 1010_2, where as the round quotient should have treated this case as being between $0.10_2 * 2^{-2}$ and $0.11_2 * 2^{-2}$, and rounded to the nearest 2 rightmost bits to produce the correct *subnormal* mantissa of 011_2; however, we instead rounded (incorrectly) down to $0.010 * 2^{-2} = 0.00625$.

2.4.2.1 Modified Algorithm

The benefits of performing a rounded quotient that results in a number with *rep_mantissa_length* bits for cases when the binary result will be normalized are the following:

1. We compute the correct normalized mantissa of the result, which can be scaled by simply modifying the exponent rather than further modifying the mantissa
2. We maintain information needed to shift the final result to the correct exponent value after the mantissa has been computed.

However, neither of these apply in subnormal cases - the value we want will almost always have fewer than the number of bits in the represented mantissa, and therefore scaling is not as trivial; also, the final exponent is always known (it is 001_2 iff we rounded up to the

minimum normalized case, otherwise it is 000_2). Therefore, given that we can test if an input is above or below the minimum normalized value, we can introduce a modified version of **JAFFER-NEGATIVE** for use when cases that are below and therefore could round to subnormal results. The new version rounds directly to the final subnormal mantissa (or the represented minimum normalized mantissa) without going through a full normalized mantissa first. We will use the algorithm for all inputs that are below the minimum normalized case. Note that rather than round-quotient-scaling by the *bex* parameter (which we eliminate entirely since it is only necessary for rounding to a full mantissa), we instead use the final shift by point, and also add in a number of powers of 2 between the exact integer real represented mantissa exponent (i.e. the exponent of a binary representation of $n*2^0$ for some $n \in \{2^{stored\_mantissa\_bits}, 2^{stored\_mantissa\_bits}+1, ..., 2^{rep\_mantissa\_bits}-1\}$) and the rightmost bit of a subnormal exponent to eliminate the scaling done in the original negative algorithm (since we don't want to actually perform a round quotient yielding the correct mantissa multiplied by 2 to the power of the minimum subnormal exponent (since that would just yield 0), but rather just yielding the correct mantissa as an integer). At the end, we stick on the correct exponent based on how many bits are in the resulting mantissa.

```
algorithm JAFFER-SUBNORMAL (Integer mant, Integer point):
    // The inputs represent a value below the minimum normalized real value
    (Integer num, Integer scale) =
        QUOTIENT-SCALE-POW2 (
            mant,
            POWER (5, −point),
            point + (expBias − 1) + (stored_mantissa_length − 1)
        )
    Integer subfinal_mant = ROUND-QUOTIENT (num, scale)
    return
        if (BIT-LENGTH (subfinal_mant) == rep_mantissa_length) then
            // Have rounded up to minimum normal value
            REAL-FROM-MANT-EXP (0, 1)
        else
            REAL-FROM-MANT-EXP (subfinal_mant, 0)
```

Returning to our previous case, we trace the new algorithm in `Real7` with the same input 0.08203125 (as a reminder, *point* = −8, *mant* = 8203125):

1. `(num, scale) :=` Scale-Quotient$(8203125, 5^8, -8+(3-1)+3)$
   $=$ Scale-Quotient$(8203125, 390625, -3) = (8203125, 3125000)$

2. `subfinal_mant :=` Round-Quotient$(\texttt{num}, \texttt{scale}) =$ Round-Quotient$(8203125, 3125000)$
   $= 3$,

which is the correct mantissa, and since it is represented using $2 < 4$ bits, the exponent will remain 0, so the conversion to a real will yield the correct value of

$$(s = 0, e_s = 000_2, m_s = 3 = 011_2) = 0.09375.$$

Chapter 3

Implementation

3.1 Basics

The implementations of the algorithms discussed in the previous chapter required a basic common setup in SML, allowing all of the conversion algorithms and test-generation algorithms access to a common library. The more important features of this common library are detailed below.

3.1.1 Either.sml

A monadic Either structure, similar to the Either structure in Haskell [HSE], but offering a few more perks of SML's module system such as the ability to bind through either the `LEFT` or `RIGHT` side. This is an invaluable structure for managing the flow of data through different edge-case removals or different branches of larger and more complex algorithms, and provides a good way for the programmer to see the entirety of such algorithms in one location in a clear and concise format.

3.1.2 File Generation

The file generation library offers a small number of useful tools for writing out to files, including list writing and unified file handle structures. This is an important aspect of the project since many algorithms implemented require files to be generated for use with the compiler that are not trivial to generate by hand. MLBasis files that perform such generations when compiled and run come packaged in the appropriate places throughout the project, and either generate actual runnable structure files or important pieces of them (to be assembled

by hand if generating the full structure file is impractical for some reason) with the help of this library. However, it should be noted that all current files used in the conversions have been generated and put in place, so compilation of the project should be possible as-is without any additional generations.

3.1.3 RealExtra.sml and Related Structures

The RealExtra structures provide access to simple, unified types representing each side of the conversions. These types were chosen to closely represent the actual system types while offering more transparency and functionality on the programming side than those types. The types defined in these structures are those used by the conversion algorithms implemented. The basic types defined are as follows:

3.1.3.1 `BinaryRep.t`

This represents an IEEE-754 binary real number. The structure of the representation is simple: it stores the exponent, mantissa and sign as a `Word.word`, `LargeWord.word` and `bool`, respectively (since these types are sufficient to store the relevant portions of both the Real32.real and Real64.real). Given MLton's built-in functionality for casting back and forth between `Real<n>.real` types and the `Word<n>.word` types that represent them internally, it is trivial to extract the correct bit ranges from the real type and create data of this type, that can be more easily read and written throughout the program than a real value. The functionality for doing this is built into the RealExtra structures.

3.1.3.2 `DecimalRep.t`

This is the decimal representation of an IEEE-754 real used internally in the conversions. There are some significant differences from the basic layout of the `IEEEReal.decimal_approx` type required by the definition, which mainly deal with the handling of edge cases. The definition-required type stores digits and an exponent and sign as well as a separate piece of `IEEEReal.float_class` data, which has several important disadvantages:

1. It does not take advantage of SML's algebraic datatype system despite the fact that doing so could make the type clearer and avoid the storing of unnecessary information (e.g. an empty digit list for a NaN value)

2. It stores information that is not necessary or relevant to the decimal representation of real numbers (e.g. whether the underlying representation is normal or subnormal,

information which is not reliable and is ignored by the current native conversion algorithms when taken as input, as well as the algorithms implemented in this project)

The `DecimalRep.t` type used by the implementation does separate these pieces of data appropriately, and is declared as a `datatype`:

```
structure Digit =
    struct
        open Int
        type t = int
    end

datatype t =
    Inf of {sign: bool}
  | NaN
  | Standard of {digits: Digit.t list,
                 exp: Int.int,
                 sign: bool}
  | Zero of {sign: bool}
```

We separate zero out as well as $\pm\infty$ and `NaN` both for convenience (as the case is treated differently in the algorithms) and because it makes conversions to and from the `IEEEReal.decimal_approx` type simpler and faster (as no digits or digit lists have to be checked at all).

In order to be definition-compliant, the Binary-Decimal conversion algorithm returns a `DecimalRep.t` along with a `IEEEReal.float_class option` in order to indicate whether or not the option is normal or subnormal in the event of a standard case, but this information is only to comply with the definition: it is largely irrelevant to the conversion and is actually available as soon as the real is (trivially) converted to the internal `BinaryRep.t` before the algorithm does any work.

3.1.3.3 Opaque Typing

In order to prevent a number of degenerate cases from wandering into the algorithms, each RealExtra structure retains a `decimal_rep` and `binary_rep` type that are equal to their corresponding representations discussed above under the hood but are concealed by opaque signature bindings, accessible only through the `<rep-type>_rep_in` and `<rep-type>_rep_out` functions, which convert between the opaque and general types. The purpose of this choice was to allow for a buffer dealing with certain degenerate cases that are capable of being

removed (throwing an error) or amended (modified to be compliant) before insertion into the type with a low cost. This allows for a higher degree of safety when passing around the representations. While the outgoing function simply reveals the data hidden by the opaque type (i.e. is the identity under the hood), the incoming function does non-trivial processing. The binary function removes cases that have components too large for the various components of a real type (such as a mantissa with a 1 in the 64th bit for a 64-bit real), and amends all NaN cases into a single one that is standard for the entire project (this allows the `binary_reps` to be compared effectively as an equality type even though the reals that they represent are not). The decimal function amends representations with *leading* 0s in their digit list by pulling them off (since operations at the front of the list are cheap) and amends degenerate cases with no digits to the zero case with the same sign.

3.2 RYŪ IMPLEMENTATION

The implementation of this algorithm is reasonably straightforward. The main implementation detail to discuss is the integer storage types of the implementation's lookup tables - in the single-precision case, the lookup tables need to store 64-bit integers in all cases, whereas in the double-precision case, 128-bit integers are necessary. Although there is no native 128-bit integer type available in MLton, we would like to avoid storing infinite-precision integers in the 128-bit integer table, since this could take up a significant amount of space - beyond the 127 bits of integer (without the sign) to store, each stored value would have to store an extra GMP limb (typically 64 bits) for a single sign bit. To prevent this from happening, pairs of 64-bit unsigned words were stored and interpreted as 128-bit integers by the program. An accompanying signature was provided offering the functionality to convert back and forth between the stored type and infinite precision (where arithmetic was actually performed). In the case of the single-precision table, these functions were trivial: they simply used the MLton's built-in conversion functions to and from the infinite precision type. In the double-precision table this was more complicated: the first 64-bit word in the pair was treated as 63 bits + a sign. The sign was removed and the remaining word converted to infinite-precision and shifted to the left to allow for the other 64-bit word to be added in. The sign was then put back in place by negating the result if necessary (although no negative integers were stored, so the only sign bit encountered was 0).

3.3 JAFFER IMPLEMENTATION

The implementation of Jaffer's algorithm relies heavily on a number of important optimizations that allow for the algorithm to run smoothly without doing more infinite-precision arithmetic than is necessary. Unfortunately, unlike the Ryū algorithm, we do not seek a shortest representation of bounded size, a computation that Ulf Adams has shown can be easily optimized in order to limit the amount of computation that must be done; rather we begin with a digit list of potentially arbitrary size and must operate on it to achieve a result.

However, there are several ways in which we can restrict the input before any significant processing occurs. First of all, we can easily remove exponents that are too low or too high to be relevant. Anything with an exponent below that of the edge case between 0 and the minimum positive subnormal double can be immediately rounded to positive or negative 0. Likewise, anything with an exponent exceeding that of the edge case between the maximum positive double and infinity (which is treated as the next highest exponent would be if the IEEE-754 convention was extended to the next highest binary exponent) can be immediately rounded to positive or negative infinity.

3.3.1 Digit List Pruning

What we are left with after this removal is a reasonable range of decimal exponents with arbitrarily long digit strings. Fortunately, we can remove all but a fixed number of the digits from a given digit list given the base-10 exponent. Namely, given the base-10 exponent converted to a base-2 exponent, we know that the relevant portion of the representation has at most the same number of significant digits as the maximum mantissa (of a fixed number of bits and therefore digits) multiplied by some epsilon (which has the significant digits of a power of 2 or 5) that can be found using the base-2 exponent, plus 2 extra digits for rounding.

The conversion of a base-10 exponent to a base-2 exponent in this case is a bit nuanced because binary values with different base-2 exponents may share the same base-10 exponent. As such, we must choose the base-2 exponent that causes us to require keeping the most digits to avoid losing any information. In the end, the number of digits will be a function of the epsilon exponent, which can be defined as the base-2 exponent minus the number of bits in the mantissa (not including the implied 1) for normalized exponents and the number of significant digits in the mantissa for a given subnormal exponent, which given the minimum represented exponent e_m and the number of bits in the stored mantissa B_{m_s} is:

$$e_\epsilon = \max(e_2 - B_{m_s}, e_m - B_{m_s}).$$

We must look ahead to see what the value of this will be before making a choice from amongst the possible e_2 values to ensure that the power of 2 or 5 will have the largest possible number of digits.

We begin by computing a baseline base-2 exponent value e_{2_0} that will be a real number

$$e_{2_0} = e_{10}\log_2 10,$$

and then compute the expected corresponding epsilon exponent

$$e_{\epsilon_0} = \max(e_{2_0} - B_{m_s}, e_m - B_{m_s}).$$

Our choice of e_2 (an integer) will be affected by the sign of e_{ϵ_0}, since given the ultimate sign, we can choose our choice of rounding to ensure that the number of final digits increases. If the sign is non-negative, increasing e_2 by rounding up will cause us to take more digits. Therefore, we will compute e_2 by rounding e_{2_0} up to the nearest integer after adding an extra $\log_2 10$ (an extra digit) into the result (since e_{2_0} is really $(\log_2 10)(1.0 \times 10^{e_{10}})$, so we must add a digit to account for the case where the representation is larger, such as $9.999 \times 10^{e_{10}}$). If the sign is negative, decreasing e_2 will cause us to take more digits. Therefore, we will compute e_2 by rounding e_{2_0} down to the nearest integer.

$$e_2 = \begin{cases} \lceil e_{2_0} + \log_2 10 \rceil, & e_{\epsilon_0} \geq 0 \\ \lfloor e_{2_0} \rfloor, & e_{\epsilon_0} < 0 \end{cases}$$

We now can define e_ϵ, an integer:

$$e_\epsilon = \max(e_2 - B_{m_s}, e_m - B_{m_s}),$$

and then note the power-of-2 factor difference between e_2 and e_ϵ, as this is the number of bits in the mantissa that could be multiplied by the ϵ value:

$$e_d = e_2 - e_\epsilon.$$

Armed with these values, we can now compute the following sub-final value:

$$N_d = \begin{cases} (e_2 + e_d) * \log_{10} 2, & e_\epsilon \geq 0 \\ |e_2| * \log_{10} 5 + e_d * \log_{10} 2, & e_{\epsilon_0} < 0, \end{cases}$$

and then add 3 to N_d to arrive at the result. One digit is added for each of the following:

- The logarithm computed is one less than the number of digits we must take (since, for example, 10^{18} has 19 digits)
- The edge case between two representations could have an additional digit (e.g. if the cases 100...00 and 100...01 are reals with adjacent binary representations in a real type, the edge case between them is the real value 100...00.5).
- We could need one more digit to round correctly - if the case we are dealing with is not an edge case but is close to one, and if there are more digits to take after this digit, leaving the rest off could cause the algorithm to consider this an edge case (that could round down) rather than a non-edge case (that should always round up). To combat this issue, we take one more digit, and if that digit is 0 and there are more nonzero digits beyond this one, we change it to 1 to prevent such a case from happening. Since none of the rest of the non-zero digits are relevant (except for the fact that they exist and therefore this is not an edge case), it is acceptable to represent them all by increasing this digit if it is zero. If the digit is not zero, then it is sufficient to represent that this is not an edge case (e.g. continuing the example in the previous item, rather than truncating the digits of 100...00.50031 to 100...00.50, we would instead take 100...00.51, which should round the same as the actual input.)

3.3.2 Subnormal Case Identification

As mentioned in section 2.4.2, we need to be able to compare a decimal representation input to the minimum positive normalized case in order to determine whether to run one of the original Jaffer algorithms (if it is greater than or equal) or **JAFFER-SUBNORMAL** (if it is less). Unfortunately, the full decimal representation of the minimum value has 89 digits in single-precision, and a whopping 715 digits in double-precision - this is both a lot of digits to store and a potentially large comparison that will not be avoidable if the exponent of the input decimal representation is the same as the exponent of the minimum normalized case. However, we can avoid running the entire comparison by storing only some of the digits. We will demonstrate how using the single precision case. The minimum normalized positive value in this real type has the value $n_{\min} = 1.1754943... * 10^{-38}$. Similarly, the edge cases between this value and the adjacent representable binary reals above and below have the values $n_{\min} - (2^{-126-24} = 2^{-150}) = 1.1754942... * 10^{-38}$ and $n_{\min} + 2^{-150} = 1.1754944... * 10^{-38}$, respectively. Therefore, we know that any value with a matching exponent that starts with

the digits $1,1,7,5,4,9,4,3$ is closer to the minimum normalized value than it is to any other representable real value. Therefore, we can store only those digits. If the digit list of the input compares as less than or greater than, we have determined which Jaffer algorithm to run; if it has compared equal, we know that this case will round to $n_{\min}$, so we are done with the conversion. We make similar arrangements in the double-precision case. In the end, we can store the 8 digits of the single-precision value shown above in the single-precision case, and the 17 digits $2,2,2,5,0,7,3,8,5,8,5,0,7,2,0,1,3$ in the double-precision case.

Chapter 4

Results

4.1 Tests Generated and Correctness

A wide range of tests were performed, both for the single- and double- precision real conversions in both directions. The types of tests used were as described in section 2.2, except one additional modification for the edge case generation: after generating an edge case digit list as an infinite-precision integer, we can optionally multiply it by a power of 10 and either add or subtract 1 (but not modify the exponent), producing a case close-by to the edge case. These tests help confirm that the decimal-to-binary implementation rounds correctly nearby the edge cases on either side, and therefore that the side of the edge case that a nearby case is on determines how it rounds.

The following tests were generated for each real precision:

1. 500,000 binary representation tests (only tests that can be used for binary-to-decimal, as discussed previously)
2. 50,000 decimal edge case representation tests
3. 50,000 decimal edge case representation tests, modified as discussed above by multiplying by 10^5 and adding 1
4. 50,000 decimal edge case representation tests, modified as discussed above by multiplying by 10^5 and subtracting 1
5. 50,000 decimal exact representation tests
6. 1,000,000 decimal representations with digit lists of length 5

7. 500,000 decimal representations with digit lists of length 25
8. 200,000 decimal representations with digit lists of length 100
9. 50,000 decimal representations with digit lists of length 500
10. 50,000 decimal representations with digit lists of length 1000

The new implementations were tested for correctness, the binary-to-decimal using the 500,000 generated binary tests, and the decimal using all 2,500,000 generated tests, and successfully passed.

4.2 TIMING

Timing tests were done using tests selected from among those generated to determine how the timing of the new implementation stands up to the timing of `gtdoa`, namely the generated binary tests and the generated decimal tests that involve digit lists of fixed lengths. The tests were run on a 2016 MacBook Pro with 16GB of RAM running macOS 10.12.6 (Darwin), and with MLton version 20180207.

4.2.1 Decimal-To-Binary

Strangely, the binary-to-decimal algorithm implemented in the project was slower than the old method in the 32-bit case (taking approximately 121.7% of the time) but faster in the 64-bit case (taking approximately 69.1% of the time). It is not known why this was the case, as the algorithms for each case were identical up to a change of basic parameters passed into SML functors and an integer storage type change for the 64-bit case that should have been slower than the strategy employed in the 32-bit case.

There was an expected slowdown from 32- to 64- bits in the new implementations - the 32-bit binary tests ran in approximately 0.600s, and the 64-bit tests in approximately 0.915 seconds. This is reasonable as the 64-bit case makes use of more infinite-precision arithmetic and uses larger numbers. What is surprising is the testing for the `gtdoa` library - in the 32-bit case, the tests ran in approximately 0.493s, whereas in the 64-bit case, they took a full 1.323 seconds. It would be beneficial to account for this discrepancy, but a more in-depth analysis of the inner workings of the `gtdoa` library and MLton's use of it, and in particular how 32- and 64-bit reals are treated differently, is necessary to do so.

Bits	Number of Digits	Relative time taken (%)
32	5	205.7
32	25	208.1
32	100	169.2
32	500	249.0
32	1000	53.7
64	5	145.2
64	25	257.9
64	100	272.5
64	500	306.0
64	1000	78.4

Table 4.1: Fixed-length digit list tests - times given in percentage of the time taken to run the native implementation tests

4.2.2 Binary-To-Decimal

The binary-to-decimal algorithm implemented took more time than the old implementation, but some patterns in the slowdowns across tests suggest possible modifications that could help improve performance.

4.2.2.1 Binary Tests

When testing the 500,000 binary tests generated, the new algorithm took approximately 153.2% of the time taken by the old algorithm in the 32-bit case, and approximately 162.7% in the 64-bit case.

4.2.2.2 Fixed-Length Tests

Interesting patterns occurred with the slowdowns in the fixed-length digit list tests. In general, the slowdowns became more severe as the number of digits went up. This does not meet with expectations, because since we generate exponents randomly, there is still large-precision arithmetic required in all of the generated tests since some exponents will be extremely large or small (requiring multiplications and divisions by 5 raised to powers that are, on average, in the hundreds).

Notably, in the 64-bit case, there is a noticeable jump between the comparative time taken for 5-digit list tests and for 25-digit list tests. A possible benchmark being crossed between the two is that the infinite-precision mantissas being computed from the digit lists become large infinite-precision integers rather than the optimized small 64-bit integer versions that are required of the 5-digit list case (each small integer is stored in a GMP limb, which on the

tested architecture is a 64-bit word, in two's complement representation, leaving 63 bits to express positive integers with; one additional bit is needed internally by the runtime system, leaving 62 remaining bits. $2^{62} \approx 4.612 \times 10^{18}$ has 19 digits). Since as discussed above, the size of the arithmetic was not expected to change with the size of the digit list, this may be due to the fact that the conversion from digit list to integer is too naïve - we simply begin with 0, and multiply each number by a small power of 10, adding in a corresponding small number of digits every time. Unfortunately, this causes many non-trivial large-precision multiplications and additions to occur, and also loads the heap with intermediate large infinite-precision calculations that will not appear there if the calculations stay small.

Another interesting note is that the fixed-length digit list cases seem to be worse than the shortest-output cases for the 32-bit implementation, *even though the latter outputs are longer in some cases*. Typically, the single-precision real type has shortest information-preserving representations that are 6 or 7 digits long (since in normalized cases, the mantissa represents 24 bits, so the difference between a given representable value r and the nearest representable values is around $(1/2^{24} = 1/8,388,608)r$; however, we note that even the 5-digit fixed-length test time results are worse than the shortest-output results. This is not expected, since the lengths of the digit lists in the former tests are close to or lower than those in the latter tests. However, it is possible that this is due to trailing zeroes. The algorithms implemented rely on the removal of any trailing zeroes from the digit list input before processing takes place. Currently, the algorithm fixes digit lists passed in with trailing zeroes by reversing the list to remove them and then reversing the resulting list back, operations that are both space- and time- expensive (as list reversal takes time and allocates another list as well). Since the digit lists are randomly generated, approximately 1 in 10 of the tests will have trailing 0s, whereas none of the shortest representations will (since adding or removing trailing zeroes does not change the value that a decimal representation represents, therefore given a decimal representation with trailing zeroes in the digit list, a representation that is the same but with the trailing zeroes removed represents the same value, and is shorter. Because of this, we will always choose the latter representation over the former when finding a shortest-possible output). Thus, the slowdown could be due to the fact that these simple-but-inefficient list operations are required when running the fixed-length tests, but not the binary shortest-representation tests. In fact, recent further research suggests that this theory is correct - rather than reversing the digit list to remove trailing zeroes if they exist, we can convert the list to an array (which takes up less heap space than a list of the same size) and remove leading zeroes by creating an array slice (which takes up 0 extra heap space). Implementing this change drastically lowered the time taken for the single-precision 5-digit fixed-length decimal

representation test cases (which it was theorized were most affected by the inefficiency) from around 205% of the time taken by the original implementation to around 120%.

Looking at the 1000-digit cases, we see that the new implementation is suddenly faster than `gtdoa`, a likely cause being that the previous implementation does not end up doing any optimizations of the digit list. Namely, neither the single-precision nor the double-precision cases ever require this many digits to round correctly (with the double-precision taking a little over 700 digits at the most), so our algorithm can peel off the unneeded digits in all of these cases, whereas the current implementation does whatever necessary arithmetic it requires with the full representation passed in.

Chapter 5

Final Words

5.1 Conclusion

We have implemented new algorithms for standard ML that correctly round in performing binary-to-decimal and decimal-to-binary conversions. These implementations are in native MLton code, and so are capable of being integrated into the compiler in order to replace the current external `gtdoa` library as the built-in implementations of these conversions. Furthermore, the timing gains in the 64-bit case of integrating our Ryū implementation to the compiler outweigh the losses in the 32-bit case.

5.2 Future Work

There is more work to be done before the `gtdoa` library can be fully replaced by the implementation. First (and most importantly), the other decimal problems discussed in this report, including the fixed-length output problem, are addressed by the external library (and required by the SML definition); therefore, replacing the library requires coming up with a native replacement for this functionality.

The timing issues for the decimal-to-binary case and the binary-to-decimal should be investigated. The analysis of our results suggest two important first steps into this work:

1. Coming up with a more efficient method of converting from digit lists to integers could prevent us from having to compute excessive numbers of large integers and store them in the heap

2. Finding a more efficient way of working with the digit list operations could prevent list reversals and re-allocations. Performing a conversion to a vector or array if necessary

may be a good alternative, as it would allow for the list to be shortened quickly (by using a slice) while also taking up less space in the heap than another list. Some research that has been done in this area has proved promising, as noted in the previous chapter.

Finally, the implementation has yet to be integrated into MLton, which is the obvious final step. Besides building the functionality into MLton's real structures, this may involve some modifications to the existing structure of the project, such as moving relevant utilities into MLton's libraries where they can be exposed to programmers *as well as* for use in the project, rather than maintaining all tools in one location for use in the implementations only. Some tools such as `Either.sml` could be useful as library packages for applications that lie outside the scope of this project. `MLton.Real32` and `MLton.Real64` might also benefit from some of the tools on the `RealExtra` structures.

Bibliography

[Ada18] U. Adams. Ryū: Fast float-to-string conversion. 2018.

[Gay] D. Gay. gtdoa library. `http://www.netlib.org/fp/gdtoa.tgz`.

[Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23:5–48, 1991.

[HSE] Haskell `Either` documentation. `http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Either.html`.

[Jaf15] A. Jaffer. Easy accurate reading and writing of floating-point numbers. 2015.