Universität zu Köln

Mathematisches Institut

Bachelorarbeit

The Unum Number Format: Mathematical Foundations, Implementation and Comparison to IEEE 754 Floating-Point Numbers

Laslo Hunhold

Erstgutachterin:
Prof. Dr. Angela Kunoth

Zweitgutachter:
Samuel Leweke

8. November 2016

Contents

1. Introduction

This thesis examines a modern concept for machine numbers based on interval arithmetic called ‘Unums’ and compares it to IEEE 754 floating-point arithmetic, evaluating possible uses of this format where floating-point numbers are inadequate. In the course of this examination, this thesis builds theoretical foundations for IEEE 754 floating-point numbers, interval arithmetic based on the projectively extended real numbers and Unums.

Machine Number Concepts Following the invention of machine floating-point numbers by Leonardo Torres y Quevedo in 1913 (see [Ran82, Section 3]) the format has evolved to be the standard used in numerical computations today. Over time though, different concepts and new approaches for representing numbers in a machine have emerged. One of these new concepts is the *infinity computer* developed by Yaroslav D. Sergeyev, introducing *grossone* arithmetic for superfinite calculations. See [Ser15] for further reading.

Another concept are the *universal numbers* (‘Unums’) proposed by John L. Gustafson. They were first introduced as a variable-length floating-point format with an uncertainty-bit as a superset of IEEE 754 floating-point numbers called ‘Unums 1.0’ (see [Gus15]). Reasoning about the complexity of machine implementations for and decimal calculation trade-offs with IEEE 754 floating-point numbers (see [Gus16b, Section 2]), Gustafson presented a new format aiming to be easy to implement in the machine and provide a simple way to do decimal calculations with guaranteed error bounds (see [Gus16a] and [Gus16b]). He called the new format ‘Unums 2.0’. In the course of this thesis, we are referring to ‘Unums 2.0’ when talking about Unums.

Projectively Extended Real Numbers Besides the well-known and established concept of extending the real numbers with signed infinities $+\infty$ and $-\infty$, called the *affinely extended real numbers*, a different approach is to only use one unsigned symbol for infinity, denoted as $\breve{\infty}$ in this thesis. This extension is called the *projectively extended real numbers* and we will prove that it is well-defined in terms of finite and infinite limits. It is in our interest to examine how much we lose and what we gain with this reduction, especially in regard to interval arithmetic.

Interval Arithmetic The concept behind interval arithmetic is to model quantities bounded by two values, thus in general being subsets rather than elements of the real numbers. Despite the fact that interval arithmetic in the machine can give definite bounds for a result, it is easy to find examples where it gives overly pessimistic results, for instance the dependency problem.

This thesis will present a theory of interval arithmetic based on the projectively extended real numbers, picking up the idea of modelling degenerate intervals across the infinity point as well, allowing division by zero and showing many other useful properties.

Goal of this Thesis The goal of this thesis is to evaluate the Unum number format in a theoretical and practical context, make out advantages and see how reasonable it is to use Unums rather than the ubiquitous IEEE 754 floating-point format for certain tasks.

At the time of writing, all available implementations of the Unum arithmetic are using floating-point arithmetic at runtime instead of solely relying on lookup tables as GUSTAFSON proposes. The provided toolbox developed in the course of this thesis limits the use of floating-point arithmetic at runtime to the initialisation of input data. Thus it is a special point of interest to evaluate the format the way it was proposed and not in an artificial floating-point environment created by the currently available implementations.

Structure of this Thesis Following Chapter 2, which provides a formalisation of IEEE 754 floating-point numbers from the ground up solely based on the standard, deriving characteristics of the set of floating-point numbers and using numerical examples that show weaknesses of the format, Section 3.1 introduces the projectively extended real numbers and proves well-definedness of this extension after introducing a new concept of finite and infinite limits on it. Based on this foundation, Sections 3.2 and 3.3 construct a theory of interval arithmetic on top of the projectively extended real numbers, formalizing intuitive concepts of interval arithmetic.

Using the results obtained in Chapter 3, Chapter 4 embeds the Unums 2.0 number format proposed by John L. GUSTAFSON (see [Gus16a] and [Gus16b]) within this interval arithmetic, evaluating it both from a theoretical and practical perspective, providing a Unum 2.0 toolbox that was developed in the course of this thesis and giving numerical examples implemented in this toolbox.

2. IEEE 754 Floating-Point Arithmetic

Floating-point numbers have gone a long way since Konrad ZUSE's Z1 and Z3, which were among the first machines to implement floating-point numbers, back then obviously using a non-standardised format (see [Roj98, pp. 31, 40–48]). With more and more computers seeing the light of day in the decades following the pioneering days, the demand for a binary floating-point standard rose in the face of many different proprietary floating-point formats.

The Institute of Electrical and Electronics Engineers (IEEE) took on the task and formulated the 'ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic' (see [IEE85]), published and adopted internationally in 1985 and revised in 2008 (see [IEE08]) with a few extensions, including decimal floating-point numbers (see [IEE08, Section 3.5]), which are not going to be presented here. This standardisation effort led to a homogenisation of floating-point formats across computer manufacturers, and this chapter will only deal with this standardised format and follow the concepts presented in the IEE 754-2008 standard. All results in this chapter are solely derived from this standard.

2.1. Number Model

The idea behind floating-point numbers rests on the observation that given a base $b \in \mathbb{N}$ with $b \geq 2$ any $x \in \mathbb{R}$ can be represented by

$$\exists (s, e, d) \in \{0,1\} \times \mathbb{Z} \times \{0, \ldots, b-1\}^{\mathbb{N}_0} : x = (-1)^s \cdot b^e \cdot \sum_{i=0}^{\infty} d_i \cdot b^{-i}.$$

There exist multiple parametres (s, e, d) for a single x. For instance, $x = 6$ in the base $b = 10$ yields $(0, 0, \{6, 0, \ldots\})$ and $(0, 1, \{0, 6, 0, \ldots\})$ as two of many possible parametrisations.

Given the finite nature of the computer, the number of possible exponents e and digits d_i is limited. Within these bounds we can model a machine number $\tilde{x}$ with exponent bounds $\underline{e}, \overline{e} \in \mathbb{Z}$, $\underline{e} \leq e \leq \overline{e}$ and a fixed number of digits $n_m \in \mathbb{N}$ and base $b = 2$ as

$$\tilde{x} = (-1)^s \cdot 2^e \cdot \sum_{i=0}^{n_m} d_i \cdot 2^{-i}.$$

Given binary is the native base the computer works with, we will assume $b = 2$ in this chapter. Despite being able to model finite floating-point numbers in the machine now, we still have problems with the lack of uniqueness. The IEEE 754 standard solves this by reminding that the only difference between those multiple parametrisations for a given machine number $\tilde{x} \neq 0$ is that

$$\min \{i \in \{0, \ldots, n_m\} \mid d_i = 0\}$$

is variable (see [IEE08, Section 3.4]). This means that we have a varying amount of 0's in the sequence $\{d_i\}_{i \in \{0,\ldots,n_m\}}$ until we reach the first 1. One way to work around this redundancy is to use *normal* floating point numbers, which force $d_0 = 1$ (see [IEE08, Section 3.4]). The d_0 is not stored as it has always the same value. This results in the

Definition 2.1 (set of normal floating-point numbers)**.** *Let $n_m \in \mathbb{N}$ and $\underline{e}, \overline{e} \in \mathbb{Z}$. The set of normal floating-point numbers is defined as*

$$\mathbb{M}_1(n_m, \underline{e}, \overline{e}) := \left\{ (-1)^s \cdot 2^e \cdot \left(1 + \sum_{i=1}^{n_m} d_i \cdot 2^{-i} \right) \;\middle|\; s \in \{0,1\} \wedge \underline{e} \leq e \leq \overline{e} \wedge d \in \{0,1\}^{n_m} \right\}.$$

In addition to normal floating-point numbers, we can also define *subnormal* floating-point numbers, also known as *denormal* floating-point numbers, which force $d_0 = 0$ and $e = \underline{e}$ and are smaller in magnitude than the smallest (positive) normal floating-point number (see [IEE08, Section 3.4d]).

Definition 2.2 (set of subnormal floating-point numbers)**.** *Let $n_m \in \mathbb{N}$ and $\underline{e} \in \mathbb{Z}$. The set of subnormal floating-point numbers is defined as*

$$\mathbb{M}_0(n_m, \underline{e}) := \left\{ (-1)^s \cdot 2^{\underline{e}} \cdot \left(0 + \sum_{i=1}^{n_m} d_i \cdot 2^{-i} \right) \;\middle|\; s \in \{0,1\} \wedge d \in \{0,1\}^{n_m} \right\}.$$

The subnormal floating-point numbers allow us to express 0 with $d = 0$ and fill the so called 'underflow gap' between the smallest normal floating-point number and 0. With d and s variable, we use boundary values of the exponent to fit subnormal, normal and exception cases under one roof (see [IEE08, Section 3.4a-e]).

Definition 2.3 (set of floating-point numbers)**.** *Let $n_m \in \mathbb{N}$, $\underline{e}, \overline{e} \in \mathbb{Z}$ and $d \in \{0,1\}^{n_m}$. The set of floating point numbers is defined as*

$$\mathbb{M}(n_m, \underline{e} - 1, \overline{e} + 1) \ni: \tilde{x}(s, e, d) \begin{cases} \in \mathbb{M}_0(n_m, \underline{e}) & e = \underline{e} - 1 \\ \in \mathbb{M}_1(n_m, \underline{e}, \overline{e}) & \underline{e} \leq e \leq \overline{e} \\ = (-1)^s \cdot \infty & e = \overline{e} + 1 \wedge d = 0 \\ = \mathrm{NaN} & e = \overline{e} + 1 \wedge d \neq 0. \end{cases}$$

In the interest of comparing different parametrisations for $\mathbb{M}$, we want to find expressions for the smallest positive non-zero subnormal, smallest positive normal and largest normal floating-point numbers.

Proposition 2.4 (smallest positive non-zero subnormal floating-point number)**.** *Let $n_m \in \mathbb{N}$ and $\underline{e} \in \mathbb{Z}$. The smallest positive non-zero floating-point number is*

$$\min \left(\mathbb{M}_0(n_m, \underline{e}) \cap \mathbb{R}^+_{\neq 0} \right) = 2^{\underline{e} - n_m}.$$

Proof. Let $0 \neq d \in \{0,1\}^{n_m}$. It follows that

$$\min\left(\mathbb{M}_0(n_m,\underline{e}) \cap \mathbb{R}^+_{\neq 0}\right) = \min\left((-1)^0 \cdot 2^{\underline{e}} \cdot \left[0 + \sum_{i=1}^{n_m} d_i \cdot 2^{-i}\right]\right) = 2^{\underline{e}} \cdot 2^{-n_m} = 2^{\underline{e}-n_m}. \quad \square$$

Proposition 2.5 (smallest positive normal floating-point number)**.** *Let* $n_m \in \mathbb{N}$ *and* $\underline{e}, \overline{e} \in \mathbb{Z}$*. The smallest positive normal floating-point number is*

$$\min\left(\mathbb{M}_1(n_m,\underline{e},\overline{e}) \cap \mathbb{R}^+_{\neq 0}\right) = 2^{\underline{e}}.$$

Proof. Let $0 \neq d \in \{0,1\}^{n_m}$ and $\underline{e} \leq e \leq \overline{e}$. It follows that

$$\min\left(\mathbb{M}_1(n_m,\underline{e},\overline{e}) \cap \mathbb{R}^+_{\neq 0}\right) = \min\left((-1)^0 \cdot 2^{e} \cdot \left[1 + \sum_{i=1}^{n_m} d_i \cdot 2^{-i}\right]\right) = 2^{\underline{e}}. \quad \square$$

Proposition 2.6 (largest normal floating-point number)**.** *Let* $n_m \in \mathbb{N}$ *and* $\underline{e}, \overline{e} \in \mathbb{Z}$*. The largest normal floating-point number is*

$$\max\left(\mathbb{M}_1(n_m,\underline{e},\overline{e})\right) = 2^{\overline{e}} \cdot \left(2 - 2^{-n_m}\right).$$

Proof. Let $d \in \{0,1\}^{n_m}$ and $\underline{e} \leq e \leq \overline{e}$. It follows with the finite geometric series that

$$\begin{aligned}
\max\left(\mathbb{M}_1(n_m,\underline{e},\overline{e})\right) &= \max\left((-1)^s \cdot 2^{e} \cdot \left[1 + \sum_{i=1}^{n_m} d_i \cdot 2^{-i}\right]\right) \\
&= (-1)^0 \cdot 2^{\overline{e}} \cdot \left(1 + \sum_{i=1}^{n_m} 2^{-i}\right) \\
&= 2^{\overline{e}} \cdot \sum_{i=0}^{n_m} 2^{-i} \\
&= 2^{\overline{e}} \cdot \sum_{i=0}^{n_m} \left(\frac{1}{2}\right)^i \\
&= 2^{\overline{e}} \cdot \frac{1 - \left(\frac{1}{2}\right)^{n_m+1}}{1 - \frac{1}{2}} \\
&= 2^{\overline{e}} \cdot \left(2 - 2^{-n_m}\right) \qquad \square
\end{aligned}$$

Proposition 2.7 (number of NaN representations)**.** *Let* $n_m \in \mathbb{N}$ *and* $\underline{e}, \overline{e} \in \mathbb{Z}$*. The number of* NaN *representations is*

$$|\,\mathrm{NaN}\,|(n_m) := \left|\left\{\tilde{x}(s,e,d) \in \mathbb{M}(n_m,\underline{e}-1,\overline{e}+1) \,\middle|\, \tilde{x} = \mathrm{NaN}\right\}\right| = 2^{n_m+1} - 2.$$

Proof. Let $0 \neq d \in \{0,1\}^{n_m}$. It follows from Defintion 2.3 that

$$\tilde{x}(s,e,d) = \mathrm{NaN} \quad \Leftrightarrow \quad e = \overline{e} + 1 \wedge d \neq 0.$$

This means that there are $2^{n_m} - 1$ possible choices for d, yielding with the arbitrary $s \in \{0,1\}$ that

$$|\,\mathrm{NaN}\,|(n_m) = 2 \cdot (2^{n_m} - 1) = 2^{n_m+1} - 2. \qquad \square$$

2.2. Memory Structure

It is in our interest to map $\mathbb{M}(n_m, \underline{e} - 1, \overline{e} + 1)$ into a memory region, more specifically a bit array. The format defined by the IEEE 754-2008 standard is shown in Figure 2.1, where n_e stands for the number of bits in the exponent, n_m for the bits in the mantissa and the leading single bit is reserved for the sign bit.

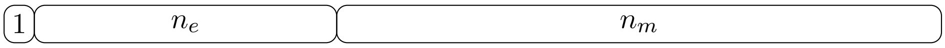

Figure 2.1.: IEEE 754 Floating-point memory layout; see [IEE08, Figure 3.1].

Handling the exponent just as an unsigned integer would not allow the use of negative exponents. To solve this, the so called *exponent bias* was introduced in the IEEE 754 standard, which is the value $2^{n_e-1}-1$ subtracted from the unsigned value of the exponent (see [IEE08, Section 3.4b]) and should not be confused with the *two's complement*, the usual way to express signed integers in a machine. Looking at the exponent values, the exponent bias results in

$$\underline{e} - 1 = -2^{n_e-1} + 1 \leq e \leq 2^{n_e} - 2^{n_e-1} = \overline{e} + 1$$

and thus

$$(\underline{e}, \overline{e}) = (-2^{n_e-1} + 2, 2^{n_e} - 2^{n_e-1} - 1) = (-2^{n_e-1} + 2, 2^{n_e-1} - 1).$$

This can be formally expressed as the

Definition 2.8 (exponent bias)**.** *Let $n_e \in \mathbb{N}$. The exponent bias is defined as*

$$\begin{aligned}\underline{e}(n_e) &:= -2^{n_e-1} + 2\\ \overline{e}(n_e) &:= 2^{n_e-1} - 1.\end{aligned}$$

With the exponent bias representation, we know how many exponent values can be assumed. Because of that it is now possible to determine the

Proposition 2.9 (number of normal floating-point numbers)**.** *Let $n_m, n_e \in \mathbb{N}$. The number of normal floating-point numbers is*

$$|\mathbb{M}_1(n_m, \underline{e}(n_e), \overline{e}(n_e))| = 2^{1+n_e+n_m} - 2^{n_m+2}.$$

Proof. According to Definition 2.1 there are

$$\overline{e}(n_e) - \underline{e}(n_e) + 1 = 2^{n_e-1} - 1 + 2^{n_e-1} - 2 + 1 = 2^{n_e} - 2$$

different exponents for $\mathbb{M}_1(n_m, \underline{e}(n_e), \overline{e}(n_e))$. Given $d \in \{0,1\}^{n_m}$ and $s \in \{0,1\}$ are arbitrary it follows that

$$|\mathbb{M}_1(n_m, \underline{e}(n_e), \overline{e}(n_e))| = 2 \cdot 2^{n_m} \cdot (2^{n_e} - 2) = 2^{1+n_e+n_m} - 2^{n_m+2}. \qquad \square$$

Proposition 2.10 (number of subnormal floating-point numbers)**.** *Let* $n_m, n_e \in \mathbb{N}$. *The number of subnormal floating-point numbers is*

$$|\mathbb{M}_0(n_m, \underline{e}(n_e))| = 2^{n_m+1}.$$

Proof. According to Definition 2.2 it follows with arbitrary $d \in \{0,1\}^{n_m}$ and $s \in \{0,1\}$ that

$$|\mathbb{M}_0(n_m, \underline{e}(n_e))| = 2 \cdot 2^{n_m} = 2^{n_m+1}. \qquad \square$$

Proposition 2.11 (number of floating-point numbers)**.** *Let* $n_m, n_e \in \mathbb{N}$. *The number of floating point numbers is*

$$|\mathbb{M}(n_m, \underline{e}(n_e)+1, \overline{e}(n_e)-1)| = 2^{1+n_e+n_m}.$$

Proof. We define

$$|\infty| := \left| \left\{ \tilde{x}(s,e,d) \in \mathbb{M}(n_m, \underline{e}-1, \overline{e}+1) \,\middle|\, \tilde{x} = \pm\infty \right\} \right| = 2$$

and conclude from Definition 2.3 that

$$\begin{aligned} |\mathbb{M}(n_m, \underline{e}(n_e)+1, \overline{e}(n_e)-1)| &= |\mathbb{M}_0(n_m, \underline{e}(n_e))| + |\mathbb{M}_1(n_m, \underline{e}(n_e), \overline{e}(n_e))| + |\infty| + |\operatorname{NaN}| \\ &= 2^{n_m+1} + 2^{1+n_e+n_m} - 2^{n_m+2} + 2 + 2^{n_m+1} - 2 \\ &= 2^{1+n_e+n_m} + 2^{n_m+1} + 2^{n_m+1} - 2 \cdot 2^{n_m+1} \\ &= 2^{1+n_e+n_m}. \end{aligned}$$

$\square$

Excluding the extended precisions above 64 bit, the IEEE 754 standard defines three storage sizes for floating-point numbers (see [IEE08, Section 3.6]), parametrised by n_m and n_e, as can be seen in Table 2.1. Half precision floating-point numbers (binary16) were introduced in IEEE 754-2008 and are just meant to be a storage format and not used for arithmetic operations given the low dynamic range.

2.3. Rounding

Given $\mathbb{M}(n_m, \underline{e}-1, \overline{e}+1)$ is a finite set, we need a way to map arbitrary real values into it if we want floating-point numbers to be a useful model of the real numbers. The IEEE 754 standard solves this with *rounding*, an operation mapping real values to preferrably close floating-point numbers based on a set of rules (see [IEE08, Section 4.3]). Given the different requirements depending on the task at hand, the IEEE 754 standard defines five rounding rules. Two based on rounding to the nearest value (see [IEE08, Section 4.3.1]) and three based on a directed approach (see [IEE08, Section 4.3.2]).

precision	half (binary16)	single (binary32)	double (binary64)
storage size (bit)	16	32	64
n_e (bit)	5	8	11
n_m (bit)	10	23	52
exponent bias	15	127	1023
$\underline{e}$	-14	-126	-1022
$\overline{e}$	15	127	1023
$\min(\mathbb{M}_0 \cap \mathbb{R}^+_{\neq 0})$	$\approx 5.96 \cdot 10^{-8}$	$\approx 1.40 \cdot 10^{-45}$	$\approx 4.94 \cdot 10^{-324}$
$\min(\mathbb{M}_1 \cap \mathbb{R}^+_{\neq 0})$	$\approx 6.10 \cdot 10^{-5}$	$\approx 1.18 \cdot 10^{-38}$	$\approx 2.23 \cdot 10^{-308}$
$\max(\mathbb{M}_1)$	$\approx 6.55 \cdot 10^{+4}$	$\approx 3.40 \cdot 10^{+38}$	$\approx 1.80 \cdot 10^{+308}$
$\lvert\mathbb{M}_0\rvert$	$\approx 2.04 \cdot 10^{+3}$	$\approx 1.68 \cdot 10^{+7}$	$\approx 9.01 \cdot 10^{+15}$
$\lvert\mathbb{M}_1\rvert$	$\approx 6.14 \cdot 10^{+4}$	$\approx 4.26 \cdot 10^{+9}$	$\approx 1.84 \cdot 10^{+19}$
$\lvert \mathrm{NaN} \rvert$	$\approx 2.05 \cdot 10^{+3}$	$\approx 1.68 \cdot 10^{+7}$	$\approx 9.01 \cdot 10^{+15}$
$\lvert\mathbb{M}\rvert$	$\approx 6.55 \cdot 10^{+4}$	$\approx 4.29 \cdot 10^{+9}$	$\approx 1.84 \cdot 10^{+19}$
$\lvert \mathrm{NaN} \rvert / \lvert\mathbb{M}\rvert$ (%)	≈ 3.12	≈ 0.39	≈ 0.05

Table 2.1.: IEEE 754-2008 binary floating-point numbers up to 64 bit with their characterizing properties.

2.3.1. Nearest

The most intuïtive approach is to just round to the nearest floating-point number. In case of a tie though, there has to be a rule in place to make a decision possible. Two rules proposed by the IEEE 754 standard are *tiing to even* (also known as *Banker's rounding*) and *tiing away from zero*. Only the first mode is presented here, which is also the default rounding mode (see [IEE08, Section 4.3.3]).

This part of the standard is often misunderstood, resulting in many publications not presenting nearest and tie to even rounding as the standard rounding operation but nearest and tie away from zero rounding, which is not correct but easy to overlook.

Definition 2.12 (nearest and tie to even rounding)**.** *Let* $n_m \in \mathbb{N}$*,* $\underline{e}, \overline{e} \in \mathbb{Z}$ *and* $x \in \mathbb{R}$ *with* $(s, e, d) \in \{0,1\} \times \mathbb{Z} \times \{0,1\}^{\mathbb{N}_0}$ *satisfying*

$$x = (-1)^s \cdot 2^e \cdot \sum_{i=0}^{\infty} \left(d_i \cdot 2^{-i} \right).$$

The nearest and tie to even rounding reduction

$$\mathrm{rd}_{\mathcal{E}} \colon \mathbb{R} \to \mathbb{M}(n_m, \underline{e} - 1, \overline{e} + 1)$$

is defined for

$$\begin{aligned}
\underline{x} &:= (-1)^s \cdot 2^e \cdot \sum_{i=0}^{n_m} \left(d_i \cdot 2^{-i}\right) \\
\overline{x} &:= (-1)^s \cdot 2^e \cdot \left[\sum_{i=0}^{n_m} \left(d_i \cdot 2^{-i}\right) + 1 \cdot 2^{-n_m}\right]
\end{aligned}$$

as

$$x \mapsto \begin{cases} (-1)^s \cdot \infty & |x| \geq \max(\mathbb{M}_1) - \left(2^{\overline{e}} \cdot 2^{-n_m}\right) = 2^{\overline{e}} \cdot \left(2 - 2^{(-n_m-1)}\right) \\ \overline{x} & |x - \overline{x}| < |x - \underline{x}| \vee \left[|x - \overline{x}| = |x - \underline{x}| \wedge d_{n_m} = 1\right] \\ \underline{x} & |x - \overline{x}| > |x - \underline{x}| \vee \left[|x - \overline{x}| = |x - \underline{x}| \wedge d_{n_m} = 0\right]. \end{cases}$$

What this means is that if two nearest machine numbers $\underline{x}$ and $\overline{x}$ are equally close to x, the last mantissa bit d_{n_m} of $\underline{x}$ decides whether x is rounded to $\underline{x}$ or $\overline{x}$. For $d_{n_m} = 0$ we know that $\underline{x}$ is even and for $d_{n_m} = 1$ it follows from the definition that $\overline{x}$ is even.

Tiing to even may seem like an arbitrary and complicated approach to rounding, but its stochastic properties make it very useful to avoid biased rounding-effects in only one direction. Given for a set of rounding-operations the number of even and odd ties, if they appear, will be roughly the same with the number of rounding-operations approaching infinity, it results in a balanced behaviour of up- and downrounding in tie-cases.

2.3.2. Directed

Another way to round numbers is a directed rounding approach to a given orientation. The three modes have three distinct orientations: Rounding *toward zero*, *upward* and *downward*. The first mode is not presented here.

Definition 2.13 (upward rounding)**.** *Let* $n_m \in \mathbb{N}$, $\underline{e}, \overline{e} \in \mathbb{Z}$ *and* $x \in \mathbb{R}$ *with* $(s, e, d) \in \{0,1\} \times \mathbb{Z} \times \{0,1\}^{\mathbb{N}_0}$ *satisfying*

$$x = (-1)^s \cdot 2^e \cdot \sum_{i=0}^{\infty} \left(d_i \cdot 2^{-i}\right).$$

The upward rounding reduction

$$\mathrm{rd}_{\uparrow} \colon \mathbb{R} \to \mathbb{M}(n_m, \underline{e} - 1, \overline{e} + 1)$$

is defined for $\underline{x}, \overline{x}$ *as in Definition 2.12 as*

$$x \mapsto \begin{cases} \underline{x} & x < 0 \\ \begin{cases} \underline{x} & \forall i > n_m : d_i = 0 \\ \overline{x} & \exists i > n_m : d_i = 1 \end{cases} & x \geq 0. \end{cases}$$

Definition 2.14 (downward rounding)**.** *Let* $n_m \in \mathbb{N}$, $\underline{e}, \overline{e} \in \mathbb{Z}$ *and* $x \in \mathbb{R}$ *with* $(s, e, d) \in \{0,1\} \times \mathbb{Z} \times \{0,1\}^{\mathbb{N}_0}$ *satisfying*

$$x = (-1)^s \cdot 2^e \cdot \sum_{i=0}^{\infty} \left(d_i \cdot 2^{-i} \right).$$

The downward rounding reduction

$$\mathrm{rd}_{\downarrow} \colon \mathbb{R} \to \mathbb{M}(n_m, \underline{e} - 1, \overline{e} + 1)$$

is defined for $\underline{x}, \overline{x}$ *as in Definition 2.12 as*

$$x \mapsto \begin{cases} \begin{cases} \underline{x} & \forall i > n_m : d_i = 0 \\ \overline{x} & \exists i > n_m : d_i = 1 \end{cases} & x < 0 \\ \underline{x} & x \geq 0. \end{cases}$$

The directed rounding modes are important for interval-arithmetic where it is important not to round down the upper bound or round up the lower bound of an interval. This way it is always guaranteed that for $a, b \in \mathbb{R}$ and $a \leq b$

$$[a, b] \subseteq [\mathrm{rd}_{\downarrow}(a), \mathrm{rd}_{\uparrow}(b)] \tag{2.1}$$

is satisfied. The bounds may grow faster than by using a to-nearest rounding mode, but it is guaranteed that the solution lies inbetween them.

2.4. Problems

As with any numerical system, we can find problems exhibiting its weaknesses. In this context we examine three different kinds of problems. Using the results obtained here it will allow us to evaluate if and how good the Unum arithmetic solves these problems respectively.

2.4.1. The Silent Spike

This example has been taken from [Kah06, §7] and simplified. Consider the function $f \colon \mathbb{R} \to \mathbb{R}$ defined as

$$f(x) := \ln(|3 \cdot (1 - x) + 1|). \tag{2.2}$$

It is easy to see that we hit a spike where

$$\begin{aligned} & |3 \cdot (1 - x) + 1| = 0 \\ \Leftrightarrow\quad & 3 \cdot (1 - x) + 1 = 0 \\ \Leftrightarrow\quad & 3 - 3 \cdot x + 1 = 0 \\ \Leftrightarrow\quad & x = \frac{4}{3}. \end{aligned}$$

More specifically,

$$\lim_{x \downarrow \frac{4}{3}} (f(x)) = \lim_{x \uparrow \frac{4}{3}} (f(x)) = -\infty.$$

Implementing this problem using IEE 754 floating-point numbers (see listing B.1.1), we might expect to receive a very small number or even negative infinity in an environment of $\frac{4}{3}$. However, this is not the case.

Instead, as you can see in Figure 2.2, the program claims that $f(\frac{4}{3}) \approx -36.044$ is the minimum in direct vicinity of $\frac{4}{3}$, completely hiding the fact that f is singular in $\frac{4}{3}$. The reason why the floating-point implementation hides the singularity is not that the logarithm implementation is faulty, but because the value passed to the logarithm is off in the first place. It is easy to see the singular point $\frac{4}{3}$ cannot be exactly represented in the machine. This effect is increased with rounding errors occuring during the evaluation (see Listing B.1.1) of

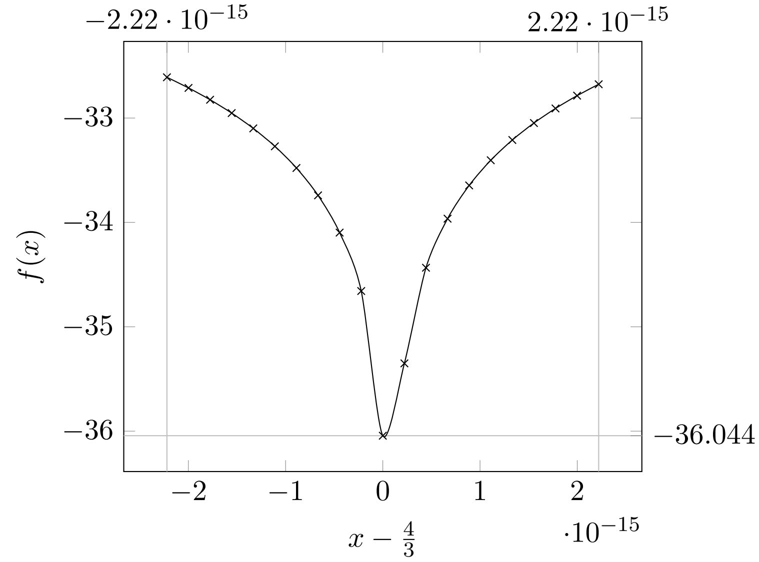

Figure 2.2.: Interpolated evaluations (demarked by crosses) of f (see (2.2)) in the neighbourhood of $\frac{4}{3}$ for all possible double floating-point numbers in $\left[\frac{4}{3} - 2.22 \cdot 10^{-15}, \frac{4}{3} + 2.22 \cdot 10^{-15}\right]$ (see listing B.1.1).

$$\left| \mathrm{rd}_{\mathcal{E}} \left\{ \mathrm{rd}_{\mathcal{E}} \left[\mathrm{rd}_{\mathcal{E}}(3) \cdot \mathrm{rd}_{\mathcal{E}} \left(\mathrm{rd}_{\mathcal{E}}(1) - \mathrm{rd}_{\mathcal{E}} \left(\frac{4}{3} \right) \right) \right] + \mathrm{rd}_{\mathcal{E}}(1) \right\} \right| \approx 2.2204 \cdot 10^{-16}.$$

In magnitude, this is relatively close to zero, but given

$$\ln(2.2204 \cdot 10^{-16}) \approx -36.0437$$

we not only see the significance of the rounding error, but also the reason why the floating-point implementation claims that -36.044 is the minimum of f in direct vicinity of $\frac{4}{3}$.

This result indicates that there are simple examples where floating-point numbers fail for piecewise continuous functions with singularities. Not being able to spot singularities for a given function might have drastic consequences, for example ‘hiding’ destructive frequencies in resonance curves for the oscillation of bridge stay cables, which are, for instance, derived in [PdCMBL96].

2.4.2. Devil’s Sequence

This example has been taken from [MBdD$^+$10, Chapter 1.3.2]. Consider the recurrent series $\{u_n\}_{n \in \mathbb{N}_0}$ defined as

$$u_n := \begin{cases} 2 & n = 0 \\ -4 & n = 1 \\ 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1} \cdot u_{n-2}} & n \geq 2 \end{cases} \tag{2.3}$$

and determine the possible limits of this series, if they exist. For this purpose, we assume convergence with $u := u_n = u_{n-1} = u_{n-2}$ and obtain the characteristic polynomial relation

$$\begin{aligned} & u = 111 - \frac{1130}{u} + \frac{3000}{u^2} \\ \Leftrightarrow \quad & u^3 = 111 \cdot u^2 - 1130 \cdot u + 3000 \\ \Leftrightarrow \quad & 0 = u^3 - 111 \cdot u^2 + 1130 \cdot u - 3000 \end{aligned}$$

with solutions 5, 6 and 100. As further described in [Kah06, §5] for a similar recurrence, we obtain the general solution with $\alpha, \beta, \gamma \in \mathbb{R}$ under the condition $|\alpha| + |\beta| + |\gamma| \neq 0$

$$u_n = \frac{\alpha \cdot 100^{n+1} + \beta \cdot 6^{n+1} + \gamma \cdot 5^{n+1}}{\alpha \cdot 100^n + \beta \cdot 6^n + \gamma \cdot 5^n}. \tag{2.4}$$

For $u_0 = 2$ and $u_1 = -4$ we obtain $\alpha = 0$ and $\gamma = -\frac{3}{4} \cdot \beta \neq 0$, resulting in

$$\begin{aligned} u_n &= \frac{6^{n+1} - \frac{3}{4} \cdot 5^{n+1}}{6^n - \frac{3}{4} \cdot 5^n} \\ &= \frac{6^{n+1} - \frac{3}{4} \cdot \left(\frac{5}{6} \cdot 6\right)^{n+1}}{6^n - \frac{3}{4} \cdot \left(\frac{5}{6} \cdot 6\right)^n} \\ &= \frac{6^{n+1}}{6^n} \cdot \frac{1 - \frac{3}{4} \cdot \left(\frac{5}{6}\right)^{n+1}}{1 - \frac{3}{4} \cdot \left(\frac{5}{6}\right)^n} \\ &= 6 \cdot \frac{1 - \frac{3}{4} \cdot \left(\frac{5}{6}\right)^{n+1}}{1 - \frac{3}{4} \cdot \left(\frac{5}{6}\right)^n}. \end{aligned}$$

It follows that

$$\lim_{n \to \infty} (u_n) = 6.$$

If we take a look at the floating-point implementation (see listing B.1.2) of this problem, we can observe a rather peculiar behaviour: Figure 2.3 shows that the IEEE 754-based solver behaves completely opposite from what one might expect. Using the closed form (2.4) we have shown that the recurrence (2.3) converges to 6. However, even though the floating-point solver comes quite close to 6 up until $n = 15$, it unexpectedly converges to 100 in subsequent iterations. The reason for that is found within consecutive rounding errors of u_n, which skew the results so far that the parametre α of the closed form (2.4) becomes non-zero.

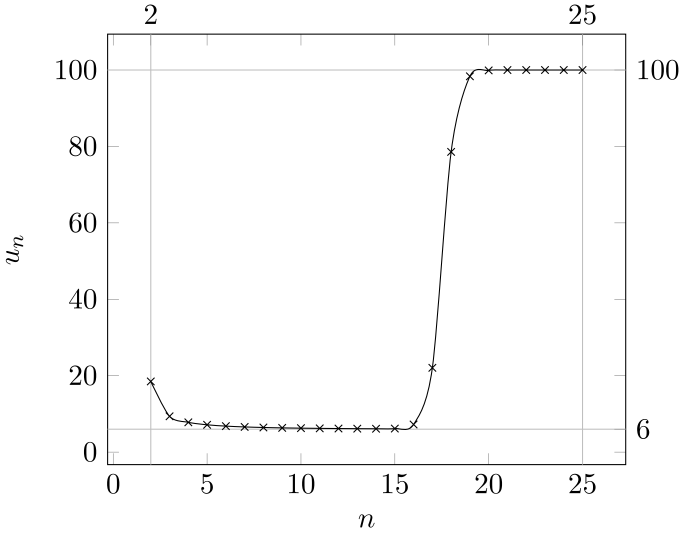

Figure 2.3.: Interpolated double floating-point evaluations (demarked by crosses) of the devil's sequence u_n (see (2.3)) for $n \in \{2, \dots, 25\}$ (see listing B.1.2).

The carefully chosen starting values $u_0 = 2$ and $u_1 = -4$ deliberately make α disappear in (2.4), which shows how even little rounding errors can give completely wrong results for such a pathologic example.

2.4.3. The Chaotic Bank Society

This example has been taken from [MBdD+10, Chapter 1.3.2]. Consider the recurrent series $\{a_n\}_{n \in \mathbb{N}_0}$ defined for $a_0 \in \mathbb{R}$ as

$$a_n := \begin{cases} a_0 & n = 0 \\ a_{n-1} \cdot n - 1 & n \geq 1 \end{cases} \tag{2.5}$$

with the task being to determine u_{25} for $a_0 = e - 1$.

The name of this example can be derived by thinking of the series as an imaginary offer by a bank to start with a deposit of $e - 1$ currency units and in each year for 25 years, multiply it by the current running year number and subtract one currency unit as banking charges.

For a theoretical answer, we first want to find a closed form of u_n. We observe the pattern

$$\begin{aligned}
a_0 &= a_0 && = 0! \cdot (a_0) \\
a_1 &= a_0 \cdot 1 - 1 && = 1! \cdot \left(a_0 - \frac{1}{1!}\right) \\
a_2 &= (a_0 \cdot 1 - 1) \cdot 2 - 1 && = 2! \cdot \left(a_0 - \frac{1}{1!} - \frac{1}{2!}\right) \\
a_3 &= [(a_0 \cdot 1 - 1) \cdot 2 - 1] \cdot 3 - 1 && = 3! \cdot \left(a_0 - \frac{1}{1!} - \frac{1}{2!} - \frac{1}{3!}\right).
\end{aligned}$$

This leads us to the

Proposition 2.15 (closed form of a_n)**.** *The closed form of the recurrent series (2.5) is*

$$a_n = n! \cdot \left(a_0 - \sum_{k=1}^{n} \frac{1}{k!}\right)$$

Proof. We prove the statement by induction over $n \in \mathbb{N}_0$.

a) $a_0 = a_0 = 0! \cdot a_0$.

b) Assume $a_n = n! \cdot \left(a_0 - \sum_{k=1}^{n} \frac{1}{k!}\right)$ holds true for an arbitrary but fixed $n \in \mathbb{N}$.

c) Show $n \mapsto n + 1$.

$$\begin{aligned}
a_{n+1} &= a_n \cdot (n+1) - 1 \\
&\overset{b)}{=} n! \cdot \left(a_0 - \sum_{k=1}^{n} \frac{1}{k!}\right) \cdot (n+1) - 1 \\
&= (n+1)! \cdot \left(a_0 - \sum_{k=1}^{n} \frac{1}{k!} - \frac{1}{(n+1)!}\right) \\
&= (n+1)! \cdot \left(a_0 - \sum_{k=1}^{n+1} \frac{1}{k!}\right)
\end{aligned}$$

□

Using the closed form of a_n and the definition of Euler's number, we get for a disturbed $a_0 = (e - 1) + \delta$ with $\delta \in \mathbb{R}$

$$\begin{aligned}
a_n &= n! \cdot \left((e - 1) + \delta - \sum_{k=1}^{n} \frac{1}{k!} \right) \\
&= n! \cdot \left(\delta + e - 1 - \sum_{k=1}^{n} \frac{1}{k!} \right) \\
&= n! \cdot \left(\delta + \sum_{k=0}^{+\infty} \frac{1}{k!} - \sum_{k=0}^{n} \frac{1}{k!} \right) \\
&= n! \cdot \left(\delta + \sum_{k=n+1}^{+\infty} \frac{1}{k!} \right) \\
&= n! \cdot \delta + \sum_{k=n+1}^{+\infty} \frac{n!}{k!}.
\end{aligned}$$

It follows that

$$\lim_{n \to +\infty} (a_n) = \begin{cases} -\infty & \delta < 0 \\ 0 & \delta = 0 \\ +\infty & \delta > 0 \end{cases}$$

and, thus, we can assume $a_{25} \in (0, e - 1)$ for an undisturbed $a_0 = e - 1$. In regard to the banking context this means that this offer would not be favourable for any investor.

A sloppy but quicker approach to get an answer to the problem is to write a program based on IEEE 754 floating-point numbers to calculate the account balance a_{25} (see listing B.1.3). However, the answer it gives is $a_{25} = 1201807247.410449$, suggesting a profitable offer by the bank, which it clearly is not. The reason for this erratic behaviour is that

$$\mathrm{rd}_{\mathcal{E}}(1.7182818284590452 35) > e - 1,$$

resulting in $\delta > 0$ and a_n going towards positive infinity.

This example shows how rounding errors in floating-point arithmetic can lead to false predictions and ultimately decisions, indicating the need for guaranteed solution bounds. As elaborated in Subsection 2.3.1, the nearest and tie to even rounding reduction has some advantages, but in cases like this can skew the result undesiredly and unexpectedly due to the inhomogenous behaviour of rounding. Because of that, using another constant expression for a value close to $e - 1$ might result in the answer going towards negative infinity.

3. Interval Arithmetic

The foundation for modern interval arithmetic was set by Ramon E. Moore in 1967 (see [Moo67]) as a means for automatic error analysis in algorithms. Since then, the usage of interval arithmetic beyond stability analysis was limited to some applications (see [MKŠ$^+$06], [Moo79] and [MKC09]), which is also indicated by the fact that the first IEEE standard for interval arithmetic, IEEE 1788-2015, was published in 2015 (see [IEE15]). The standard is based on the ubiquitous *affinely extended real numbers*

$$\overline{\mathbb{R}} := \mathbb{R} \cup \{+\infty\} \cup \{-\infty\},$$

which this chapter will not make use of. Instead, the basis will be the *projectively extended real numbers*

$$\mathbb{R}^* := \mathbb{R} \cup \{\breve{\infty}\}.$$

The motivation for this chapter is to find out how much we lose when only having one symbol for infinity, and more importantly, what we gain in this process, ultimately proving well-definedness of $\mathbb{R}^*$. Based on the findings, it is in our interest to construct an interval arithmetic on top of $\mathbb{R}^*$, which we can later use to formalise the Unum arithmetic.

3.1. Projectively Extended Real Numbers

With respect to simple reciprocation and negation of numbers, the projectively extended real numbers come to mind. Topologically speaking, this is the Alexandroff compactification of $\mathbb{R}$ with the point $\breve{\infty} \notin \mathbb{R}$ (see [Kow14, Section 25.4] for further reading).

As one can see in Figure 3.1, the geometric projection of $\mathbb{R}$ and infinity $\breve{\infty}$ onto a circle, and thinking of reciprocation and negation as horizontal and vertical reflections on this circle respectively, is the ideal model in this context, presenting an intuïtive approach to arithmetic operations on sets of real numbers.

Just like we can not definitely give the number 0 a sign and just by convention denote it as a positive number, there is no reason for its reciprocal $\breve{\infty}$ to have a sign. As intuïtive as this approach is, rigorous results and a formal definition are necessary to build a solid foundation for interval arithmetic on the projectively extended real numbers. In the course of the following chapter we are going to define finite and infinite limits on the projectively extended real numbers and show well-definedness of this extension in terms of infinite limits. The formal definition of $\mathbb{R}^*$ is according to [Rei82].

Definition 3.1 (projectively extended real numbers)**.** *The projectively extended real numbers are defined as*

$$\mathbb{R}^* := \mathbb{R} \cup \{\breve{\infty}\}.$$

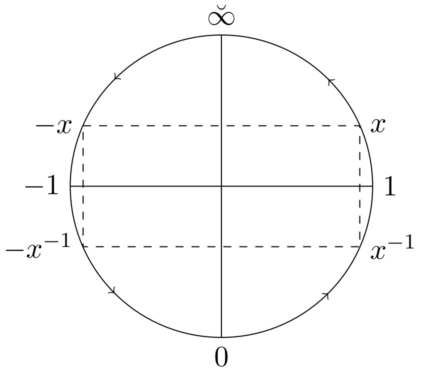

Figure 3.1.: Schema of $\mathbb{R}^*$ with the counter-clockwise orientation indicated by arrows (see Definition 3.1).

The arithmetic operations $+$ *and* $\cdot$ *are partially extended for* $a, b \in \mathbb{R}$ *with* $b \neq 0$ *to*

$$\begin{aligned} -(\breve{\infty}) &:= \breve{\infty} && \text{(3.1a)} \\ a + \breve{\infty} = \breve{\infty} + a &:= \breve{\infty} && \text{(3.1b)} \\ b \cdot \breve{\infty} = \breve{\infty} \cdot b &:= \breve{\infty} && \text{(3.1c)} \\ a/\breve{\infty} &:= 0 && \text{(3.1d)} \\ b/0 &:= \breve{\infty}. && \text{(3.1e)} \end{aligned}$$

Left undefined are $\breve{\infty} + \breve{\infty}$*,* $\breve{\infty} \cdot \breve{\infty}$*,* $0 \cdot \breve{\infty}$*,* $0/0$*,* $\breve{\infty}/\breve{\infty}$ *and* $\breve{\infty}/0$*.*

For more information on indeterminate forms on extensions of the real numbers see [TF95].

To be able to show well-definedness of the extension of the arithmetic operations in $\mathbb{R}^*$ in terms of infinite limits, we first have to introduce the concept of $\breve{\infty}$-infinite limits on $\mathbb{R}^*$.

3.1.1. Finite and Infinite Limits

Since we can not use two signed symbols for infinity, namely $\pm\infty$, directed limits can be specified with the direction of approach to $\breve{\infty}$, from above or below, indicated by vertical arrows. In this regard, ascension is interpreted in regard to the natural order of $\mathbb{R}$, from smallest to largest number. Approaching $\breve{\infty}$ from below corresponds to a limit toward $+\infty$ on $\mathbb{R}$, approaching $\breve{\infty}$ from above corresponds to a limit toward $-\infty$ on $\mathbb{R}$.

There is no sacrifice in only having one symbol for infinity up to this point, given $+\infty$ and $-\infty$ can only be approached from one direction in standard analysis. Having one symbol that can be approached from two directions fills the gap seamlessly for finite limits.

Definition 3.2 ($\breve{\infty}$-finite limit)**.** *Let $f\colon \mathbb{R} \to \mathbb{R}$. The $\breve{\infty}$-finite limit of f for x approaching $\breve{\infty}$ is defined for $\ell \in \mathbb{R}$ as*

$$\begin{aligned}
\lim_{x \downarrow \breve{\infty}} (f(x)) = \ell \quad &:\Leftrightarrow \quad \forall \varepsilon > 0 : \exists c \in \mathbb{R} : \forall x \in \mathbb{R} : x < c : |f(x) - \ell| < \varepsilon \\
\lim_{x \uparrow \breve{\infty}} (f(x)) = \ell \quad &:\Leftrightarrow \quad \forall \varepsilon > 0 : \exists c \in \mathbb{R} : \forall x \in \mathbb{R} : x > c : |f(x) - \ell| < \varepsilon \\
\lim_{x \to \breve{\infty}} (f(x)) = \ell \quad &:\Leftrightarrow \quad \lim_{x \downarrow \breve{\infty}} (f(x)) = \ell \wedge \lim_{x \uparrow \breve{\infty}} (f(x)) = \ell.
\end{aligned}$$

Remark 3.3 (standard-finite limit relationship)**.** *Let $f\colon \mathbb{R} \to \mathbb{R}$ and $\ell \in \mathbb{R}$. One can convert between standard-finite limits and $\breve{\infty}$-finite limits using the relations*

$$\begin{aligned}
\lim_{x \downarrow \breve{\infty}} (f(x)) = \ell \quad &\Leftrightarrow \quad \lim_{x \to -\infty} (f(x)) = \ell \\
\lim_{x \uparrow \breve{\infty}} (f(x)) = \ell \quad &\Leftrightarrow \quad \lim_{x \to +\infty} (f(x)) = \ell.
\end{aligned}$$

Besides finite limits, we also need a way to express when a function diverges. In this regard, having only one infinity-symbol induces some losses, as only the absolute values of the functions can be evaluated. However, it still holds that if a function diverges in standard-infinite limits it also diverges in $\breve{\infty}$-infinite limits.

Definition 3.4 ($\breve{\infty}$-infinite limit)**.** *Let $f\colon \mathbb{R} \to \mathbb{R}$. The $\breve{\infty}$-infinite limit of f for $x \in \mathbb{R}$ approaching $a \in \mathbb{R}$ is defined as*

$$\begin{aligned}
\lim_{x \downarrow a} (f(x)) = \breve{\infty} \quad &:\Leftrightarrow \quad \forall \varepsilon > 0 : \exists \delta > 0 : 0 < x - a < \delta \Rightarrow |f(x)| > \varepsilon \\
\lim_{x \uparrow a} (f(x)) = \breve{\infty} \quad &:\Leftrightarrow \quad \forall \varepsilon > 0 : \exists \delta > 0 : 0 < a - x < \delta \Rightarrow |f(x)| > \varepsilon \\
\lim_{x \to a} (f(x)) = \breve{\infty} \quad &:\Leftrightarrow \quad \lim_{x \downarrow a} (f(x)) = \breve{\infty} \wedge \lim_{x \uparrow a} (f(x)) = \breve{\infty},
\end{aligned}$$

and for $x \in \mathbb{R}$ approaching $\breve{\infty}$ as

$$\begin{aligned}
\lim_{x \downarrow \breve{\infty}} (f(x)) = \breve{\infty} \quad &:\Leftrightarrow \quad \forall \varepsilon > 0 : \exists c \in \mathbb{R} : \forall x \in \mathbb{R} : x < c : |f(x)| > \varepsilon \\
\lim_{x \uparrow \breve{\infty}} (f(x)) = \breve{\infty} \quad &:\Leftrightarrow \quad \forall \varepsilon > 0 : \exists c \in \mathbb{R} : \forall x \in \mathbb{R} : x > c : |f(x)| > \varepsilon \\
\lim_{x \to \breve{\infty}} (f(x)) = \breve{\infty} \quad &:\Leftrightarrow \quad \lim_{x \downarrow \breve{\infty}} (f(x)) = \breve{\infty} \wedge \lim_{x \uparrow \breve{\infty}} (f(x)) = \breve{\infty}.
\end{aligned}$$

Remark 3.5 (standard-infinite limit relationship)**.** *Let $f\colon \mathbb{R} \to \mathbb{R}$ and $a \in \mathbb{R}$. One can convert between standard-infinite limits and $\breve{\infty}$-infinite limits using the relations*

$$\begin{aligned}
\lim_{x \downarrow a} (f(x)) = \breve{\infty} \quad &\Leftarrow \quad \lim_{x \downarrow a} (f(x)) = \pm\infty \\
\lim_{x \uparrow a} (f(x)) = \breve{\infty} \quad &\Leftarrow \quad \lim_{x \uparrow a} (f(x)) = \pm\infty \\
\lim_{x \downarrow \breve{\infty}} (f(x)) = \breve{\infty} \quad &\Leftarrow \quad \lim_{x \to -\infty} (f(x)) = \pm\infty \\
\lim_{x \uparrow \breve{\infty}} (f(x)) = \breve{\infty} \quad &\Leftarrow \quad \lim_{x \to +\infty} (f(x)) = \pm\infty.
\end{aligned}$$

3.1.2. Well-Definedness

We can now use our definitions of $\breve{\infty}$-finite and $\breve{\infty}$-infinite limits to show that $\mathbb{R}^*$ with the extensions given in Definition 3.1 is well-defined in terms of infinite limits.

Theorem 3.6 (well-definedness of $\mathbb{R}^*$)**.** *$\mathbb{R}^*$ is well-defined in terms of infinite limits.*

Proof. Let $f_{\breve{\infty}}, f_a, f_b, f_0 \colon \mathbb{R} \to \mathbb{R}$, $a, b \in \mathbb{R}$ and $b \neq 0$. Without loss of generality we assume that $\breve{\infty}$ is approached from below and specify

$$\lim_{x\uparrow\breve{\infty}} (f_{\breve{\infty}}(x)) = \breve{\infty} \tag{3.2a}$$

$$\lim_{x\uparrow\breve{\infty}} (f_a(x)) = a \tag{3.2b}$$

$$\lim_{x\uparrow\breve{\infty}} (f_b(x)) = b \tag{3.2c}$$

$$\lim_{x\uparrow\breve{\infty}} (f_0(x)) = 0. \tag{3.2d}$$

To show well-definedness, we go through each axiom given in Definition 3.1.

Let $\tilde{\varepsilon} > 0$.

(3.1a) By Definition 3.4 we know that

$$\lim_{x\uparrow\breve{\infty}} (f_{\breve{\infty}}(x)) = \breve{\infty} \quad \Leftrightarrow \quad \lim_{x\uparrow\breve{\infty}} (-f_{\breve{\infty}}(x)) = \breve{\infty}$$

and, thus, $-(\breve{\infty}) = \breve{\infty}$ is well-defined.

(3.1b) To show that $a + \breve{\infty} = \breve{\infty} + a = \breve{\infty}$ is well-defined we have to show that

$$\lim_{x\uparrow\breve{\infty}} (f_a(x) + f_{\breve{\infty}}(x)) = \lim_{x\uparrow\breve{\infty}} (f_{\breve{\infty}}(x) + f_a(x)) = \breve{\infty}. \tag{3.3}$$

Following from precondition (3.2b), Definition 3.4 and $\tilde{\varepsilon} > 0$ we know that

$$\exists c_{2,a} \in \mathbb{R} : \forall x > c_{2,a} : |f_a(x) - a| < \tilde{\varepsilon}.$$

It follows for $x > c_{2,a}$ using the reverse triangle inequality that

$$\begin{aligned} & \tilde{\varepsilon} > |f_a(x) - a| \geq ||f_a(x)| - |a|| \geq |f_a(x)| - |a| \\ \Rightarrow \quad & |f_a(x)| < \tilde{\varepsilon} + |a|. \end{aligned} \tag{3.4}$$

Following from precondition (3.2a), Definition 3.4 and $2 \cdot \tilde{\varepsilon} + |a| > 0$ we also know that

$$\exists c_{2,\breve{\infty}} \in \mathbb{R} : \forall x > c_{2,\breve{\infty}} : |f_{\breve{\infty}}(x)| > 2 \cdot \tilde{\varepsilon} + |a|. \tag{3.5}$$

Let $x > \tilde{c}_2 := \max\{c_{2,a}, c_{2,\breve{\infty}}\}$ to satisfy both (3.4) and (3.5). It follows using the reverse triangle inequality that

$$\begin{aligned} & |f_{\breve{\infty}}(x)| > 2 \cdot \tilde{\varepsilon} + |a| = \tilde{\varepsilon} + (\tilde{\varepsilon} + |a|) > \tilde{\varepsilon} + |f_a(x)| \\ \Rightarrow \quad & \tilde{\varepsilon} < |f_{\breve{\infty}}(x)| - |f_a(x)| = |f_{\breve{\infty}}(x)| - |-f_a(x)| \leq |f_{\breve{\infty}}(x) - (-f_a(x))| \\ \Rightarrow \quad & |f_a(x) + f_{\breve{\infty}}(x)| = |f_{\breve{\infty}}(x) + f_a(x)| > \tilde{\varepsilon}, \end{aligned}$$

which by Definition 3.4 is equivalent to (3.3) and was to be shown.

(3.1c) To show that $b \cdot \breve{\infty} = \breve{\infty} \cdot b = \breve{\infty}$ is well-defined we have to show that

$$\lim_{x \uparrow \breve{\infty}} \left(f_b(x) \cdot f_{\breve{\infty}}(x) \right) = \lim_{x \uparrow \breve{\infty}} \left(f_{\breve{\infty}}(x) \cdot f_b(x) \right) = \breve{\infty}. \tag{3.6}$$

Following from precondition (3.2c), Definition 3.4 and $\frac{|b|}{2} > 0$ we know

$$\exists c_{3,b} \in \mathbb{R} : \forall x > c_{3,b} : |f_b(x) - b| < \frac{|b|}{2}.$$

It follows for $x > c_{3,b}$ using the triangle and reverse triangle inequalities that

$$\begin{aligned}
& |f_b(x) - b| < \frac{|b|}{2} = \frac{|0 - b|}{2} \leq \frac{|0 - f_b(x)| + |f_b(x) - b|}{2} \\
\Rightarrow \quad & \frac{|f_b(x) - b|}{2} < \frac{|f_b(x)|}{2} \\
\Leftrightarrow \quad & |f_b(x)| > |f_b(x) - b| = |b - f_b(x)| \geq ||b| - |f_b(x)|| \geq |b| - |f_b(x)| \\
\Rightarrow \quad & |f_b(x)| > \frac{|b|}{2}.
\end{aligned} \tag{3.7}$$

Following from precondition (3.2a), Definition 3.4 and $\frac{2 \cdot \tilde{\varepsilon}}{|b|} > 0$ we also know that

$$\exists c_{3,\breve{\infty}} \in \mathbb{R} : \forall x > c_{3,\breve{\infty}} : |f_{\breve{\infty}}(x)| > \frac{2 \cdot \tilde{\varepsilon}}{|b|}. \tag{3.8}$$

Let $x > \tilde{c}_3 := \max\{c_{3,b}, c_{3,\breve{\infty}}\}$ to satisfy both (3.7) and (3.8). It follows that

$$\begin{aligned}
& |f_{\breve{\infty}}(x)| > \frac{2 \cdot \tilde{\varepsilon}}{|b|} > \frac{\tilde{\varepsilon}}{|f_b(x)|} \\
\Rightarrow \quad & |f_b(x)| \cdot |f_{\breve{\infty}}(x)| > \tilde{\varepsilon} \\
\Leftrightarrow \quad & |f_b(x) \cdot f_{\breve{\infty}}(x)| = |f_{\breve{\infty}}(x) \cdot f_b(x)| > \tilde{\varepsilon},
\end{aligned}$$

which by Definition 3.4 is equivalent to (3.6) and was to be shown.

(3.1d) To show that $a / \breve{\infty} = 0$ is well-defined we have to show that

$$\lim_{x \uparrow \breve{\infty}} \left(\frac{f_a(x)}{f_{\breve{\infty}}(x)} \right) = 0. \tag{3.9}$$

Following from precondition (3.2a), Definition 3.4 and $\frac{\tilde{\varepsilon} + |a|}{\tilde{\varepsilon}} > 0$ we know

$$\exists c_{4,\breve{\infty}} \in \mathbb{R} : \forall x > c_{4,\breve{\infty}} : |f_{\breve{\infty}}(x)| > \frac{\tilde{\varepsilon} + |a|}{\tilde{\varepsilon}}. \tag{3.10}$$

Let $x > \tilde{c}_4 := \max\{c_{2,a}, c_{4,\breve{\infty}}\}$ to satisfy both (3.4) and (3.10). It follows that

$$\begin{aligned}
& |f_a(x)| < \tilde{\varepsilon} + |a| = \tilde{\varepsilon} \cdot \frac{\tilde{\varepsilon} + |a|}{\tilde{\varepsilon}} < \tilde{\varepsilon} \cdot |f_{\breve{\infty}}(x)| \\
\Rightarrow \quad & \frac{|f_a(x)|}{|f_{\breve{\infty}}(x)|} < \tilde{\varepsilon} \\
\Leftrightarrow \quad & \left| \frac{f_a(x)}{f_{\breve{\infty}}(x)} - 0 \right| < \tilde{\varepsilon},
\end{aligned}$$

which by Definition 3.2 is equivalent to (3.9) and was to be shown.

(3.1e) To show that $b/0 = \breve{\infty}$ is well-defined we have to show that

$$\lim_{x \uparrow \breve{\infty}} \left(\frac{f_b(x)}{f_0(x)} \right) = \breve{\infty}. \tag{3.11}$$

Following from precondition (3.2a), Definition 3.4 and $\frac{|b|}{2 \cdot \tilde{\varepsilon}} > 0$ we know

$$\exists c_{5,0} \in \mathbb{R} : \forall x > c_{5,0} : |f_0(x)| < \frac{|b|}{2 \cdot \tilde{\varepsilon}} \tag{3.12}$$

Let $x > \tilde{c}_5 := \max\{c_{3,b}, c_{5,0}\}$ to satisfy both (3.7) and (3.12). It follows that

$$\begin{aligned} & |f_b(x)| > \frac{|b|}{2} = \tilde{\varepsilon} \cdot \frac{|b|}{2 \cdot \tilde{\varepsilon}} > \tilde{\varepsilon} \cdot |f_0(x)| \\ \Rightarrow \quad & |f_b(x)| > \tilde{\varepsilon} \cdot |f_0(x)| \\ \Leftrightarrow \quad & \frac{|f_b(x)|}{|f_0(x)|} > \tilde{\varepsilon} \\ \Leftrightarrow \quad & \left| \frac{f_b(x)}{f_0(x)} \right| > \tilde{\varepsilon}, \end{aligned}$$

which by Definition 3.2 is equivalent to (3.11) and was to be shown. □

3.2. Open Intervals

With well-definedness of $\mathbb{R}^*$ shown we have built a solid foundation for $\mathbb{R}^*$-interval arithmetic. Given $\mathbb{R}^*$ is not an ordered set, we have to introduce a new definition for intervals that seamlessly extend to $\breve{\infty}$. Our goal is to define operations on open intervals and singletons and to use them to model arbitrary subsets of $\mathbb{R}^*$.

To allow degenerate intervals across $\breve{\infty}$, the convention proposed in [Rei82, pp. 88-89] is to give $\mathbb{R}^*$ a counter-clockwise orientation (see Figure 3.1) and define for $\underline{a}, \overline{a} \in \mathbb{R}$ and $\underline{a} < \overline{a}$ the degenerate interval $(\overline{a}, \underline{a})$ by tracing all elements from $\overline{a}$ to $\underline{a}$. It is in our interest to formalise this intuïtive but informal approach. To denote degenerate intervals, we first need to define the

Definition 3.7 (disjoint union)**.** *Let A be a set and $\{A_i\}_{i \in I}$ a family of sets over an index set I with $A_i \subseteq A$. A is the disjoint union of $\{A_i\}_{i \in I}$, denoted by*

$$A = \bigsqcup_{i \in I} A_i,$$

if and only if

$$\forall i, j \in I : i \neq j : A_i \cap A_j = \emptyset \tag{3.13}$$

and

$$A = \bigcup_{i \in I} A_i. \tag{3.14}$$

Definition 3.8 (open $\mathbb{R}^*$-interval)**.** *Let* $\underline{a}, \overline{a} \in \mathbb{R}^*$. *An open* $\mathbb{R}^*$*-interval between* $\underline{a}$ *and* $\overline{a}$ *is defined as*

$$\mathbb{R}^* \supset (\underline{a},\overline{a}) := \begin{cases} \mathbb{R} & \underline{a}=\overline{a}=\breve{\infty}\\ \{x\in\mathbb{R}\,|\,x<\overline{a}\} & \underline{a}=\breve{\infty}\\ \{x\in\mathbb{R}\,|\,x>\underline{a}\} & \overline{a}=\breve{\infty}\\ \{x\in\mathbb{R}\,|\,\underline{a}<x<\overline{a}\} & \underline{a}\le\overline{a}\\ (\overline{a},\breve{\infty})\sqcup\{\breve{\infty}\}\sqcup(\breve{\infty},\underline{a}) & \underline{a}>\overline{a}\end{cases}$$

In the interest of defining operations on open $\mathbb{R}^*$-intervals, we introduce the

Definition 3.9 (set of open $\mathbb{R}^*$-intervals)**.** *The set of open* $\mathbb{R}^*$*-intervals is defined as*

$$\mathbb{I} := \{(\underline{a},\overline{a})\,|\,\underline{a},\overline{a}\in\mathbb{R}^*\}. \tag{3.15}$$

with the operations $\oplus\colon \mathbb{I}\times\mathbb{I}\to\mathbb{I}$ *defined as*

$$\big((\underline{a},\overline{a}),(\underline{b},\overline{b})\big) \mapsto \left\{\begin{array}{lll}
\begin{cases}\emptyset & \underline{a}\in\mathbb{R}\\ \emptyset & \underline{b},\overline{b}\in\mathbb{R}\wedge\underline{b}\ge\overline{b}\\ \mathbb{R} & \textit{else}\end{cases} & \underline{a}=\overline{a} & (3.16\text{a})\\
(\underline{b},\overline{b})\oplus(\underline{a},\overline{a}) & \underline{b}=\overline{b} & (3.16\text{b})\\
(\breve{\infty},\overline{a}+\overline{b}) & \underline{a}=\underline{b}=\breve{\infty} & (3.16\text{c})\\
(\underline{a}+\underline{b},\breve{\infty}) & \overline{a}=\overline{b}=\breve{\infty} & (3.16\text{d})\\
\mathbb{R} & \underline{a}=\overline{b}=\breve{\infty} & (3.16\text{e})\\
(\underline{b},\overline{b})\oplus(\underline{a},\overline{a}) & \overline{a}=\underline{b}=\breve{\infty} & (3.16\text{f})\\
\begin{cases}\emptyset & \underline{b}>\overline{b}\\ (\breve{\infty},\overline{a}+\overline{b}) & \textit{else}\end{cases} & \underline{a}=\breve{\infty} & (3.16\text{g})\\
\begin{cases}\emptyset & \underline{b}>\overline{b}\\ (\underline{a}+\underline{b},\breve{\infty}) & \textit{else}\end{cases} & \overline{a}=\breve{\infty} & (3.16\text{h})\\
(\underline{b},\overline{b})\oplus(\underline{a},\overline{a}) & \underline{b}=\breve{\infty} & (3.16\text{i})\\
(\underline{b},\overline{b})\oplus(\underline{a},\overline{a}) & \overline{b}=\breve{\infty} & (3.16\text{j})\\
\begin{cases}\emptyset & \underline{a}>\overline{a}\wedge\underline{b}>\overline{b}\\ (\underline{a}+\underline{b},\overline{a}+\overline{b}) & \textit{else}\end{cases} & \textit{else} & (3.16\text{k})
\end{array}\right.$$

and, using $\underline{A} := \{\underline{a}\cdot\underline{b},\underline{a}\cdot\overline{b}\}$, $\overline{A} := \{\overline{a}\cdot\underline{b},\overline{a}\cdot\overline{b}\}$ *and* $A := \underline{A}\cup\overline{A}$ *for* $\underline{a},\overline{a},\underline{b},\overline{b}\in\mathbb{R}$, $\otimes\colon\mathbb{I}\times\mathbb{I}\to\mathbb{I}$

defined as

$$\big((\underline{a},\overline{a}),(\underline{b},\overline{b})\big) \mapsto \begin{cases}
\begin{cases} \emptyset & \underline{a} \in \mathbb{R} \\ \emptyset & \underline{b},\overline{b} \in \mathbb{R} \wedge \underline{b} \geq \overline{b} \\ \mathbb{R} & \textit{else} \end{cases} & \underline{a} = \overline{a} \qquad \text{(3.17a)} \\
(\underline{b},\overline{b}) \otimes (\underline{a},\overline{a}) & \underline{b} = \overline{b} \qquad \text{(3.17b)} \\
\begin{cases} (\overline{a}\cdot\overline{b},\breve{\infty}) & \overline{a} \leq 0 \wedge \overline{b} \leq 0 \\ \mathbb{R} & \textit{else} \end{cases} & \underline{a} = \underline{b} = \breve{\infty} \qquad \text{(3.17c)} \\
\begin{cases} (\underline{a}\cdot\underline{b},\breve{\infty}) & \underline{a} \geq 0 \wedge \underline{b} \geq 0 \\ \mathbb{R} & \textit{else} \end{cases} & \overline{a} = \overline{b} = \breve{\infty} \qquad \text{(3.17d)} \\
\begin{cases} (\breve{\infty},\overline{a}\cdot\underline{b}) & \overline{a} \leq 0 \wedge \underline{b} \geq 0 \\ \mathbb{R} & \textit{else} \end{cases} & \underline{a} = \overline{b} = \breve{\infty} \qquad \text{(3.17e)} \\
(\underline{b},\overline{b}) \otimes (\underline{a},\overline{a}) & \overline{a} = \underline{b} = \breve{\infty} \qquad \text{(3.17f)} \\
\begin{cases} \mathbb{R} & \underline{b} > \overline{b} \\ (\breve{\infty},\max(\overline{A})) & \underline{b} \geq 0 \\ (\min(\overline{A}),\breve{\infty}) & \overline{b} \leq 0 \\ \mathbb{R} & \textit{else} \end{cases} & \underline{a} = \breve{\infty} \qquad \text{(3.17g)} \\
\begin{cases} \mathbb{R} & \underline{b} > \overline{b} \\ (\min(\underline{A}),\breve{\infty}) & \underline{b} \geq 0 \\ (\breve{\infty},\max(\underline{A})) & \overline{b} \leq 0 \\ \mathbb{R} & \textit{else} \end{cases} & \overline{a} = \breve{\infty} \qquad \text{(3.17h)} \\
(\underline{b},\overline{b}) \otimes (\underline{a},\overline{a}) & \underline{b} = \breve{\infty} \qquad \text{(3.17i)} \\
(\underline{b},\overline{b}) \otimes (\underline{a},\overline{a}) & \overline{b} = \breve{\infty} \qquad \text{(3.17j)} \\
\emptyset & \underline{a} > \overline{a} \wedge \underline{b} > \overline{b} \qquad \text{(3.17k)} \\
\begin{cases} (\max(\underline{A}),\min(\overline{A})) & \operatorname{sgn}(\underline{b}) = \operatorname{sgn}(\overline{b}) \\ \emptyset & \textit{else} \end{cases} & \underline{a} > \overline{a} \qquad \text{(3.17l)} \\
(\underline{b},\overline{b}) \otimes (\underline{a},\overline{a}) & \underline{b} > \overline{b} \qquad \text{(3.17m)} \\
\emptyset & \underline{a} = \overline{a} \vee \underline{b} = \overline{b} \qquad \text{(3.17n)} \\
(\min(A),\max(A)) & \textit{else} \qquad \text{(3.17o)}
\end{cases}$$

Remark 3.10 (role of empty set in definition)**.** *The use of the empty set in Definition 3.9 denotes cases where undefined behaviour occurs.*

Theorem 3.11 (well-definedness of $\mathbb{I}$)**.** $\mathbb{I}$ *is well-defined in terms of set theory.*

Proof. One can see that the operations $\oplus$ and $\otimes$ satisfy closedness with regard to $\mathbb{I}$. Symmetry is also satisfied given the explicit transposed forms (3.16b), (3.16f), (3.16i) and (3.16j) for $\oplus$ and (3.17b), (3.17f), (3.17i), (3.17j) and (3.17m) for $\otimes$.

Well-definedness in terms of set theory is based on the condition that for given $A, B \in \mathbb{I}$ the two operations $\oplus$ and $\otimes$ must satisfy

$$A \oplus B = \{a + b \,|\, a \in A \wedge b \in B\}$$

and

$$A \otimes B = \{a \cdot b \,|\, a \in A \wedge b \in B\}$$

respectively, except for cases where undefined behaviour occurs. It follows from the conditions that if either $A = \emptyset$ or $B = \emptyset$ the resulting set is also empty (see (3.16a) and (3.17a)).

Let $a, b \in \mathbb{I}$ and $\underline{a}, \overline{a}, \underline{b}, \overline{b} \in \mathbb{R}$.

(3.16a) This case either corresponds to

$$\emptyset \oplus b,$$

yielding the empy set, or

$$\mathbb{R} \oplus b,$$

yielding $\mathbb{R}$, unless b is degenerate, given it contains $\breve{\infty}$ and $\mathbb{R}^* \notin \mathbb{I}$ is undefined, or empty, yielding the empty set.

(3.16c) This case corresponds to

$$(\breve{\infty}, \overline{a}) \oplus (\breve{\infty}, \overline{b})$$

and yields, using Definition 3.8,

$$\{x \in \mathbb{R} \,|\, x < \overline{a}\} \oplus \{x \in \mathbb{R} \,|\, x < \overline{b}\} = \{x \in \mathbb{R} \,|\, x < \overline{a} + \overline{b}\} = (\breve{\infty}, \overline{a} + \overline{b}).$$

(3.16d) This case corresponds to

$$(\underline{a}, \breve{\infty}) \oplus (\overline{a}, \breve{\infty})$$

and yields, using Definition 3.8,

$$\{x \in \mathbb{R} \,|\, x > \underline{a}\} \oplus \{x \in \mathbb{R} \,|\, x > \underline{b}\} = \{x \in \mathbb{R} \,|\, x > \underline{a} + \underline{b}\} = (\underline{a} + \underline{b}, \breve{\infty}).$$

(3.16e) This case corresponds to

$$(\breve{\infty}, \overline{a}) \oplus (\underline{b}, \breve{\infty})$$

and yields, using Definition 3.8,

$$\{x \in \mathbb{R} \,|\, x < \overline{a}\} \oplus \{x \in \mathbb{R} \,|\, x > \underline{b}\} = \mathbb{R}.$$

(3.16g) This case corresponds to

$$(\breve{\infty}, \overline{a}) \oplus (\underline{b}, \overline{b})$$

and yields, using Definition 3.8, if $(\underline{b}, \overline{b})$ is degenerate

$$\{x \in \mathbb{R} \,|\, x < \overline{a}\} \oplus ((\overline{b}, \breve{\infty}) \sqcup \{\breve{\infty}\} \sqcup (\breve{\infty}, \underline{b})) = \mathbb{R}^*$$

and, thus, the empty set as $\mathbb{R}^* \notin \mathbb{I}$ is undefined, or else

$$\{x \in \mathbb{R} \,|\, x < \overline{a}\} \oplus \{x \in \mathbb{R} \,|\, \underline{b} < x < \overline{b}\} = \{x \in \mathbb{R} \,|\, x < \overline{a} + \overline{b}\} = (\breve{\infty}, \overline{a} + \overline{b}).$$

(3.16h) This case corresponds to

$$(\underline{a}, \breve{\infty}) \oplus (\underline{b}, \overline{b})$$

and yields, using Definition 3.8, if $(\underline{b}, \overline{b})$ is degenerate

$$\{x \in \mathbb{R} \,|\, x > \underline{a}\} \oplus ((\overline{b}, \breve{\infty}) \sqcup \{\breve{\infty}\} \sqcup (\breve{\infty}, \underline{b})) = \mathbb{R}^*$$

and, thus, the empty set as $\mathbb{R}^* \notin \mathbb{I}$ is undefined, or else

$$\{x \in \mathbb{R} \,|\, x > \underline{a}\} \oplus \{x \in \mathbb{R} \,|\, \underline{b} < x < \overline{b}\} = \{x \in \mathbb{R} \,|\, x > \underline{a} + \underline{b}\} = (\underline{a} + \underline{b}, \breve{\infty}).$$

(3.16k) This case corresponds to

$$(\underline{a}, \overline{a}) \oplus (\underline{b}, \overline{b})$$

and yields, using Definition 3.8, if both $(\underline{a}, \overline{a})$ and $(\underline{b}, \overline{b})$ are degenerate

$$((\overline{a}, \breve{\infty}) \sqcup \{\breve{\infty}\} \sqcup (\breve{\infty}, \underline{a})) \oplus ((\overline{b}, \breve{\infty}) \sqcup \{\breve{\infty}\} \sqcup (\breve{\infty}, \underline{b}))$$

the empty set, as $\breve{\infty} + \breve{\infty}$ is undefined. If, without loss of generality, only $(\underline{a}, \overline{a})$ is degenerate, it yields

$$((\underline{a}, \breve{\infty}) \sqcup \{\breve{\infty}\} \sqcup (\breve{\infty}, \overline{a})) \oplus (\underline{b}, \overline{b}) = (\underline{a} + \underline{b}, \breve{\infty}) \sqcup \{\breve{\infty}\} \sqcup (\breve{\infty}, \overline{a} + \overline{b}) = (\overline{a} + \overline{b}, \overline{a} + \overline{b}).$$

If neither $(\underline{a}, \overline{a})$ nor $(\underline{b}, \overline{b})$ are degenerate, it yields

$$\{x \in \mathbb{R} \,|\, \underline{a} < x < \overline{a}\} \oplus \{x \in \mathbb{R} \,|\, \underline{b} < x < \overline{b}\} = \{x \in \mathbb{R} \,|\, \underline{a} + \underline{b} < x < \overline{a} + \overline{b}\} = (\underline{a} + \underline{b}, \overline{a} + \overline{b}).$$

The cases (3.17a), (3.17c), (3.17d), (3.17e), (3.17g), (3.17h), (3.17k), (3.17l), (3.17n) and (3.17o) for $\otimes$ are shown analogously. ☐

Given the complexity of open interval arithmetic alone, it becomes clear why open intervals have been studied independently up to this point. We will now expand $\mathbb{I}$ with singletons and introduce the concept of $\mathbb{R}^*$-Flakes.

3.3. Flakes

To model subsets of $\mathbb{R}^*$, one easily finds that open intervals alone are not sufficient to model even simple sets. Using singletons to expand $\mathbb{I}$ can present new possibilities. Before we introduce the central concept of this chapter, we first need to formalise the definition of singletons in $\mathbb{R}^*$.

Definition 3.12 (set of singletons)**.** *Let S be a set. The set of S-singletons is defined as*

$$\S(S) := \{\{x\} : x \in S\}\,.$$

Now we proceed to define the expansion of $\mathbb{I}$ with $\mathbb{R}^*$-singletons as the

Definition 3.13 (set of $\mathbb{R}^*$-Flakes)**.** *Let $a, b \in \mathbb{F}$. The set of $\mathbb{R}^*$-Flakes is defined as*

$$\mathbb{F} := \mathbb{I} \sqcup \S(\mathbb{R}^*).$$

To simplify notation, set the correspondences for $\underline{a}, \overline{a}, \tilde{a}, \underline{b}, \overline{b}, \tilde{b} \in \mathbb{R}^*$

$$\begin{aligned} a \in \mathbb{I} \quad &\leftrightarrow \quad a = (\underline{a}, \overline{a}) \\ a \in \S(\mathbb{R}^*) \quad &\leftrightarrow \quad a = \{\tilde{a}\} \\ b \in \mathbb{I} \quad &\leftrightarrow \quad b = (\underline{b}, \overline{b}) \\ b \in \S(\mathbb{R}^*) \quad &\leftrightarrow \quad b = \{\tilde{b}\} \end{aligned}$$

and use them to define the operations $\boxplus \colon \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ defined as

$$(a, b) \mapsto \begin{cases} a \oplus b & a, b \in \mathbb{I} \qquad (3.18a) \\ \begin{cases} \emptyset & \tilde{a} = \tilde{b} = \breve{\infty} \\ \{\tilde{a} + \tilde{b}\} & else \end{cases} & a, b \in \S(\mathbb{R}^*) \qquad (3.18b) \\ \begin{cases} \begin{cases} \emptyset & \underline{b} \geq \overline{b} \\ \{\breve{\infty}\} & else \end{cases} & \tilde{a} = \breve{\infty} \\ \emptyset & \underline{b} = \overline{b} \\ (\tilde{a} + \underline{b}, \tilde{a} + \overline{b}) & else \end{cases} & a \in \S(\mathbb{R}^*) \wedge b \in \mathbb{I} \qquad (3.18c) \\ b \boxplus a & a \in \mathbb{I} \wedge b \in \S(\mathbb{R}^*) \qquad (3.18d) \end{cases}$$

and, using $A = \{\tilde{a} \cdot \underline{b}, \tilde{a} \cdot \overline{b}\}$ *for* $\tilde{a}, \underline{b}, \overline{b} \in \mathbb{R}$, $\boxtimes \colon \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ *defined as*

$$
(a,b) \mapsto
\begin{cases}
a \otimes b & a, b \in \mathbb{I} \qquad (3.19\text{a}) \\
\begin{cases}
\emptyset & \tilde{a} = \breve{\infty} \wedge \tilde{b} \in \{0, \breve{\infty}\} \\
\emptyset & \tilde{a} \in \{0, \breve{\infty}\} \wedge \tilde{b} = \breve{\infty} \\
\{\tilde{a} \cdot \tilde{b}\} & \textit{else}
\end{cases} & a, b \in \S(\mathbb{R}^*) \qquad (3.19\text{b}) \\
\begin{cases}
\begin{cases}
\begin{cases}
\{\breve{\infty}\} & \overline{b} < 0 \\
\emptyset & \textit{else}
\end{cases} & \underline{b} = \breve{\infty} \\
\begin{cases}
\{\breve{\infty}\} & \underline{b} > 0 \\
\emptyset & \textit{else}
\end{cases} & \overline{b} = \breve{\infty} \\
\emptyset & \underline{b} > \overline{b} \\
\{\breve{\infty}\} & \operatorname{sgn}(\underline{b}) = \operatorname{sgn}(\overline{b}) \\
\emptyset & \textit{else}
\end{cases} & \tilde{a} = \breve{\infty} \\
\begin{cases}
(\breve{\infty}, \tilde{a} \cdot \overline{b}) & \tilde{a} > 0 \\
(\tilde{a} \cdot \overline{b}, \breve{\infty}) & \tilde{a} < 0 \\
\emptyset & \textit{else}
\end{cases} & \underline{b} = \breve{\infty} \\
\begin{cases}
(\tilde{a} \cdot \underline{b}, \breve{\infty}) & \tilde{a} > 0 \\
(\breve{\infty}, \tilde{a} \cdot \underline{b}) & \tilde{a} < 0 \\
\emptyset & \textit{else}
\end{cases} & \overline{b} = \breve{\infty} \\
(\max(A), \min(A)) & \underline{b} > \overline{b} \\
\emptyset & \underline{b} = \overline{b} \\
(\min(A), \max(A)) & \textit{else}
\end{cases} & a \in \S(\mathbb{R}^*) \wedge b \in \mathbb{I} \qquad (3.19\text{c}) \\
b \boxtimes a & a \in \mathbb{I} \wedge b \in \S(\mathbb{R}^*) \qquad (3.19\text{d})
\end{cases}
$$

The inverse element of $a \in \mathbb{F}$ *for* $\boxplus$ *is defined as*

$$
-a := \begin{cases}
\{-\tilde{a}\} & a \in \S(\mathbb{R}^*) \\
\begin{cases}
\emptyset & a = \emptyset \\
(-\overline{a}, -\underline{a}) & \textit{else}
\end{cases} & a \in \mathbb{I}
\end{cases}
$$

and the inverse element of $a \in \mathbb{F}$ *for* $\boxtimes$ *is defined as*

$$
/a := \begin{cases}
\{\tilde{a}^{-1}\} & a \in \S(\mathbb{R}^*) \\
\begin{cases}
\emptyset & a = \emptyset \\
(\overline{a}^{-1}, \underline{a}^{-1}) & \textit{else}
\end{cases} & a \in \mathbb{I}.
\end{cases}
$$

While this definition is definitely complex, we can see that going step by step and first defining operations on open $\mathbb{R}^*$-intervals alone makes it easier to prove well-definedness of those operations as a whole. It shall be noted here that $\mathbb{R}^*$-Flakes allow us to model closed and open sets on $\mathbb{R}^*$ easily.

Theorem 3.14 (well-definedness of $\mathbb{F}$)**.** *$\mathbb{F}$ is well-defined in terms of set theory.*

Proof. One can see that the operations $\boxplus$ and $\boxtimes$ satisfy closedness with regard to $\mathbb{F}$. Symmetry is also satisfied given the explicit transposed forms (3.18d) for $\boxplus$ and (3.19d) for $\boxtimes$ and the fact that we have shown in Theorem 3.11 that $\oplus$ and $\otimes$ are symmetric.

Well-definedness in terms of set theory is based on the condition that for given $A, B \in \mathbb{F}$ the two operations $\boxplus$ and $\boxtimes$ must satisfy

$$A \boxplus B = \{a + b \,|\, a \in A \wedge b \in B\}$$

and

$$A \boxtimes B = \{a \cdot b \,|\, a \in A \wedge b \in B\}$$

respectively, except for cases where undefined behaviour occurs.

Let $a, b \in \mathbb{F}$ as in Definition 3.13.

(3.18a) We have shown in Proposition 3.11 that $\boxplus$ is well-defined in terms of set theory.

(3.18b) This case corresponds to

$$\{\tilde{a}\} \boxplus \{\tilde{b}\}$$

and yields

$$\{\tilde{a} + \tilde{b}\}$$

unless $\tilde{a} = \tilde{b} = \breve{\infty}$, which is undefined, where the empty set is returned.

(3.18c) This case corresponds to

$$\{\tilde{a}\} + (\underline{b}, \overline{b})$$

and yields $\{\breve{\infty}\}$ if $\tilde{a} = \breve{\infty}$ and b is not degenerate or empty, which yields the empty set. If $\tilde{a} \in \mathbb{R}$, it yields

$$(\tilde{a} + \underline{b}, \tilde{a} + \overline{b}),$$

for degenerate and non-degenerate b, unless b is empty, which yields the empty set.

The cases (3.19a), (3.19b) and (3.19c) for $\boxtimes$ are shown analogously.

What remains to be shown is that the inverse elements are well-defined. One can see that the inverse elements are all closed under $\mathbb{F}$ and map $\emptyset$ to $\emptyset$. We now have to show that the operation of an element in $\mathbb{F}$ with its respective inverse element results in a set containing the respective neutral elements of $\mathbb{R}^*$ except where undefined behaviour occurs.

For $\boxplus$ with '$-$' and $\boxtimes$ and '$/$' we observe for singletons

$$\{\tilde{a}\} \boxplus -\{\tilde{a}\} = \{\tilde{a}\} \boxplus \{-\tilde{a}\} = \{\tilde{a} - \tilde{a}\} = \{0\} \ni 0$$
$$\{\tilde{a}\} \boxtimes /\{\tilde{a}\} = \{\tilde{a}\} \boxtimes \{\tilde{a}^{-1}\} = \begin{cases} \emptyset & \tilde{a} = \breve{\infty} \\ \{1\} \ni 1 & \text{else.} \end{cases}$$

Analogously, we observe for open $\mathbb{R}^*$-intervals with $(\underline{a},\overline{a}) \neq \emptyset$

$$(\underline{a},\overline{a}) \boxplus -(\underline{a},\overline{a}) = (\underline{a},\overline{a}) \boxplus (-\overline{a},-\underline{a}) = \begin{cases} \mathbb{R} \ni 0 & \underline{a} = \breve{\infty} \vee \overline{a} = \breve{\infty} \\ \emptyset & \underline{a} > \overline{a} \\ (\underline{a}-\overline{a},\overline{a}-\underline{a}) \ni 0 & \text{else} \end{cases}$$

$$(\underline{a},\overline{a}) \boxtimes /(\underline{a},\overline{a}) = (\underline{a},\overline{a}) \boxtimes (\overline{a}^{-1},\underline{a}^{-1}) = \begin{cases} \mathbb{R} \ni 1 & \underline{a} = \breve{\infty} \vee \overline{a} = \breve{\infty} \\ \emptyset & \underline{a} > \overline{a} \\ (\frac{\underline{a}}{\overline{a}},\frac{\overline{a}}{\underline{a}}) \ni 1 & \text{else.} \end{cases}$$

It follows the well-definedness of the inverse elements. ☐

Now that we have shown well-definedness of $\mathbb{F}$, we can proceed with showing some useful properties that allow easier generalisations on Flakes. One of them is the

Definition 3.15 ($\mathbb{R}^*$-Flake evaluation of strictly increasing functions)**.** *Let $f\colon \mathbb{R} \to \mathbb{R}$ be strictly increasing. The $\mathbb{R}^*$-Flake evaluation of f*

$$f_{\mathbb{F}}\colon \mathbb{F} \to \mathbb{F}$$

is defined with the notation $f(\breve{\infty}) := \breve{\infty}$ as

$$a \mapsto \begin{cases} \{f(\tilde{a})\} & a = \{\tilde{a}\} \in \S(\mathbb{R}^*) \\ (f(\underline{a}),f(\overline{a})) & a = (\underline{a},\overline{a}) \in \mathbb{I}. \end{cases}$$

Proposition 3.16 (well-definedness of $\bullet_{\mathbb{F}}$)**.** *The $\mathbb{R}^*$-Flake evaluation of strictly increasing functions is well-defined in terms of set theory.*

Proof. Let $f\colon \mathbb{R} \to \mathbb{R}$ be strictly increasing. We can see that $f_{\mathbb{F}}$ is closed in $\mathbb{F}$ and maps $\emptyset$ to $\emptyset$. For singletons well-definedness follows immediately, as it just corresponds to the singleton of the single function evaluation of f. In this context, $f(\breve{\infty}) = \breve{\infty}$, treating $\breve{\infty}$ as an invariant object, is also consistent with the axioms of Definition 3.1, as

$$\lim_{x\uparrow\breve{\infty}}(f(x)) = \lim_{x\downarrow\breve{\infty}}(f(x)) = \breve{\infty}.$$

For non-degenerate non-empty open $\mathbb{R}^*$-intervals the bounds grow accordingly, as

$$\forall \underline{a},\overline{a} \in \mathbb{R}: \quad \underline{a} < \overline{a} \quad \Leftrightarrow \quad f(\underline{a}) < f(\overline{a}).$$

This also implies the well-definedness of the degenerate case, as for $\underline{a},\overline{a} \in \mathbb{R}$ and $\underline{a} > \overline{a}$ it holds that

$$\begin{aligned} f_{\mathbb{F}}((\underline{a},\overline{a})) &= f_{\mathbb{F}}((\overline{a},\breve{\infty})) \sqcup f_{\mathbb{F}}(\{\breve{\infty}\}) \sqcup f_{\mathbb{F}}((\breve{\infty},\underline{a})) \\ &= (f_{\mathbb{F}}(\overline{a}),\breve{\infty}) \sqcup \{\breve{\infty}\} \sqcup (\breve{\infty},f_{\mathbb{F}}(\underline{a})) \\ &= (f_{\mathbb{F}}(\underline{a}),f_{\mathbb{F}}(\overline{a})). \end{aligned}$$ ☐

Definition 3.17 ($\mathbb{R}^*$ Flake evaluation of strictly decreasing functions)**.** *Let* $f\colon \mathbb{R} \to \mathbb{R}$ *be strictly decreasing. The* $\mathbb{R}^*$ *Flake evaluation of* f

$$f_{\mathbb{F}}\colon \mathbb{F} \to \mathbb{F}$$

is defined as

$$a \mapsto -((-f)_{\mathbb{F}}(a)).$$

With these results we have shown in general that we can evaluate strictly monotonic functions on $\mathbb{R}^*$-Flakes, for instance exp or ln confined to $\mathbb{R}^+_{\neq 0}$, which will be used later. We require strictly monotonic functions, as a constant function $f(x) = c \in \mathbb{R}$, that is monotonic but not strictly monotonic, would yield

$$f_{\mathbb{F}}((1,2)) = (f(1), f(2)) = (c,c) = \emptyset,$$

which is not well-defined in terms of set theory.

Using the results obtained in this Chapter, we can now examine a discrete set of Unums as a subset of $\mathbb{F}$. This especially allows us to use those now well-defined operations and identify them on the set of Unums, provided we choose it properly.

4. Unum Arithmetic

This Chapter will construct the Unum arithmetic based on the results in Chapter 3 and the publications [Gus16a] and [Gus16b] by GUSTAFSON. We start off by examining the

Definition 4.1 (set of Unums)**.** *Let*

$$P = \{p_1, \dots, p_n \,|\, \forall i < j : p_i < p_j\} \subset (1, \breve{\infty}),$$

$p_0 := 1$ *and* $p_{n+1} := \breve{\infty}$*. The set of Unums on the lattice* P *is defined as*

$$\begin{aligned}\mathbb{F} \supset \mathbb{U}(P) :=& \bigsqcup_{i=1}^{n} \left[\{p_i\} \sqcup /\{p_i\} \sqcup -\{p_i\} \sqcup -/\{p_i\}\right] \sqcup \\ & \bigsqcup_{i=0}^{n} \left[\{(p_i, p_{i+1})\} \sqcup \{/(p_i, p_{i+1})\} \sqcup \{-(p_i, p_{i+1})\} \sqcup \{-/(p_i, p_{i+1})\}\right] \sqcup \\ & \{1\} \sqcup \{-1\} \sqcup \{0\} \sqcup \{\breve{\infty}\}\end{aligned}$$

Remark 4.2. *By Definition 4.1,* $\mathbb{U}$ *is closed under inversion with regard to* $\boxplus$ *and* $\boxtimes$*.*

In regard to $\mathbb{F}$, Remark 4.2 underlines the fact that this choice for $\mathbb{U}$, generated by a set of lattice points between $(1, \breve{\infty})$, is in fact a good one. We will now proceed to derive some elemental properties of $\mathbb{U}$ and prepare it to define operations on it.

Proposition 4.3 (cardinality of $\mathbb{U}$)**.** *Let* P *as in Definition 4.1. The number of Unums is*

$$|\mathbb{U}| = 8 \cdot (|P| + 1).$$

Proof. Each quadrant of $\mathbb{R}^*$ is filled with $|P|$ lattice points and $|P| + 1$ intervals. Added to this are the 4 fixed points 1, -1, 0, $\breve{\infty}$. It follows from Definition 4.1 of $|\mathbb{U}|$ as a disjoint union of finite sets that

$$|\mathbb{U}| = 4 \cdot |P| + 4 \cdot (|P| + 1) + 4 = 4 \cdot (2 \cdot |P| + 2) = 8 \cdot (|P| + 1). \qquad \square$$

Before we proceed with constructing operations on the set of Unums, we first have to define the

Definition 4.4 (power set)**.** *Let* S *be a set. The power set of* S *is defined as*

$$\mathcal{P}(S) := \{s \subseteq S\}.$$

To use the results we have derived for $\mathbb{F}$, we need to find a way to 'blur' $\mathbb{R}^*$-Flakes into sets of Unums. For this purpose, we define the

Definition 4.5 (blur operator). *Let P as in Definition 4.1. The blur operator*

$$\mathrm{bl}\colon \mathbb{F} \to \mathcal{P}(\mathbb{U}(P))$$

is defined as

$$f \mapsto \{u \in \mathbb{U} : f \subseteq u\}.$$

We are now able to embed $\mathbb{R}^*$-Flakes into subsets of $\mathbb{U}$, which allows us to define operations on $\mathbb{U}$ by identifying them with operations on $\mathbb{F}$ using the bl-operator.

Remark 4.6 (dependent sets and dependency problem). *It is not within the scope of this thesis to elaborate on the theory of dependent sets, and there are multiple ways to approach it. To give a simple example, evaluating for $A = (-1, 1) \in \mathbb{I}$*

$$A - A$$

is expected to yield $\{0\}$, but using interval arithmetic, the expression just decays to

$$(-1, 1) - (-1, 1) = (-1, 1) + (-1, 1) = (-2, 2),$$

effectively doubling the width of the interval. This is known as the dependency problem.

It is in our interest to find an approach to limit this problem. As follows, we will denote two dependent sets S_1 and S_2 with $S_1 \sim S_2$, and with regard to the example given above, it holds that $A \sim A$.

To approach the dependency problem, we only evaluate pairwise operations for dependent sets. The underlying idea is that if a given value is present in the first set within a Unum, the dependency guarantees it will also only be within this Unum in the second set. We identify operations on $\mathbb{F}$ with operations on $\mathbb{U}$ by defining the

Definition 4.7 (dual Unum operation). *Let $\star\colon \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ be an operation on $\mathbb{F}$ and P as in Definition 4.1. The dual Unum operation*

$$\langle\star\rangle\colon \mathcal{P}(\mathbb{U}(P)) \times \mathcal{P}(\mathbb{U}(P)) \to \mathcal{P}(\mathbb{U}(P))$$

is defined as

$$(U, V) \mapsto \bigcup_{u \in U} \bigcup_{v \in V} \begin{cases} \emptyset & U \sim V \wedge u \neq v \\ \mathbb{R}^* & u \star v = \emptyset \\ \mathrm{bl}(u \star v) & \textit{else.} \end{cases}$$

Remark 4.8 (NaN for Unum operations). *As one can see in Definition 4.7, when an $\mathbb{R}^*$-Flake operation $\star$ yields the empty set, indicating an empty set or that undefined behaviour was witnessed, the Unum arithmetic proposed by* Gustafson *in [Gus16b, Table 2] mandates that the respective dual Unum operation yields $\mathbb{R}^*$.*

This is not the ideal behaviour, as we carefully defined $\boxplus$ and $\boxtimes$ to give the empty set if one operand is the empty set, $-\emptyset = \emptyset$ and $/\emptyset = \emptyset$. This behaviour is useful, as just like NaN *for floating-point numbers, which, once it occurs, is carried through the entire stream of floating-point calculations, the empty set plays this special role in the Unum context.*

In the interest of staying compatible with the Unum format proposed by Gustafson, *this weak spot in the proposal was implemented in the Unum toolbox anyway.*

Definition 4.9. *(Unum evaluation of strictly increasing functions) Let* $f\colon \mathbb{R} \to \mathbb{R}$ *be strictly increasing. The Unum evaluation of* f

$$\langle f_{\mathbb{F}} \rangle \colon \mathcal{P}(\mathbb{U}(P)) \to \mathcal{P}(\mathbb{U}(P))$$

is defined as

$$U \mapsto \bigcup_{u \in U} \mathrm{bl}(f_{\mathbb{F}}(u)).$$

Definition 4.10 (Unum evaluation of strictly decreasing functions)**.** *Let* $f\colon \mathbb{R} \to \mathbb{R}$ *be strictly decreasing. The Unum evaluation of* f

$$\langle f_{\mathbb{F}} \rangle \colon \mathcal{P}(\mathbb{U}(P)) \to \mathcal{P}(\mathbb{U}(P))$$

is defined as

$$U \mapsto \bigcup_{u \in U} \mathrm{bl}(-((-f)_{\mathbb{F}}(u))).$$

4.1. Lattice Selection

Until now, we have worked with arbitrary P. This set of lattice points is the only parametrisation for $\mathbb{U}$, so we want to investigate what the ideal construction of P is.

4.1.1. Linear Lattice

The simplest approach is a linear distribution of p lattice points up to a maximum value $m \in (1, \breve{\infty})$.

Definition 4.11 (linear Unum lattice)**.** *Let* $p \in \mathbb{N}$ *and* $m \in (1, \breve{\infty})$*. The linear Unum lattice with* p *lattice points and maximum* m *is defined as*

$$P_L(p, m) := \left\{ p_i := 1 + i \cdot \frac{m-1}{p} \,\middle|\, i \in \{1, \dots, p\} \right\}.$$

Proposition 4.12 (well-definedness of the linear Unum lattice)**.** *Let* $p \in \mathbb{N}$ *and* $m \in (1, \breve{\infty})$*.* $P_L(p, m)$ *is well-defined in terms of Definition 4.1.*

Proof. The desired properties $|P_L(p, m)| = p$ and $\max(P_L(p, m)) = m$ follow from Definition 4.11. We show that

$$\forall i > j : p_i > p_j.$$

This is given because $m - 1 > 0$ and

$$p_i - p_j = (i - j) \cdot \frac{m-1}{p} > 0.$$

The proof is finished by showing that $p_i \in (1, \breve{\infty})$. It suffices to prove that $p_1, p_p \in (1, \breve{\infty})$, as $\forall i > j : p_i > p_j$ and the boundary points dictate the behaviour of the interior points.

$$\begin{aligned} p_1 &= 1 + \frac{m-1}{p} \in (1, \breve{\infty}) \\ p_p &= 1 + m - 1 = m \in (1, \breve{\infty}) \end{aligned}$$

□

The problem with a linear Unum lattice is the lack of dynamic range. Just like with floating-point numbers, we want a dense distribution of lattice points around 1 and a lighter distribution the further we move away from 1. As we can deduce from this observation, a desired quality of the Unum lattice could be, for instance, an exponential distribution.

4.1.2. Exponential Lattice

Definition 4.13 (exponential Unum lattice)**.** *Let $p \in \mathbb{N}$ and $m \in (1, \breve{\infty})$. The exponential Unum lattice with p lattice points and maximum m is defined as*

$$P_E(p, m) := \left\{ p_i := \exp\left(i \cdot \frac{\ln(m)}{p} \right) \,\middle|\, i \in \{1, \dots, p\} \right\}.$$

Proposition 4.14 (well-definedness of the exponential Unum lattice)**.** *Let $p \in \mathbb{N}$ and $m \in (1, \breve{\infty})$. $P_E(p, m)$ is well-defined in terms of Definition 4.1.*

Proof. The desired properties $|P_E(p, m)| = p$ and $\max(P_E(p, m)) = m$ follow from Definition 4.13. We show that

$$\forall i > j : p_i > p_j.$$

This is given because exp is strictly monotonically increasing and

$$p_i - p_j = \exp\left(i \cdot \frac{\ln(m)}{p} \right) - \exp\left(j \cdot \frac{\ln(m)}{p} \right) > 0.$$

The proof is finished by showing that $p_i \in (1, \breve{\infty})$.

$$p_i = \exp\left(i \cdot \frac{\ln(m)}{p} \right) > \exp(0) = 1$$ □

The problem of an exponential Unum lattice is that the lattice points may have an ideal distribution, but fall onto rather inaccessible points. For such a number system to work, it has to contain a decent amount of integers, which is not the case here.

4.1.3. Decade Lattice

A different approach is to specify the number of desired significant decimal digits of each lattice point and fill the set by scaling with multiples of 10. For example, specifying 1 significant digit yields

$$P = \{2, 3, \dots, 9, 10, 20, 30, \dots, 90, 100, 200, 300, \dots\}.$$

We define this formally, using the remainder of the EUCLIDean division of a by b, denoted by $a \bmod b$ for $a \in \mathbb{N}_0$ and $b \in \mathbb{N}$, as the

Definition 4.15 (decade Unum lattice)**.** *Let $p \in \mathbb{N}_0$ and $s \in \mathbb{N}$. The decade Unum lattice with p lattice points and s significant digits is defined as*

$$P_D(p, s) := \left\{ p_i := \left[1 + 10^{-(s-1)} \cdot \left(i \bmod (10^s - 10^{s-1}) \right) \right] \cdot 10^{\left\lfloor \frac{i}{10^s - 10^{s-1}} \right\rfloor} \,\middle|\, i \in \{1, \dots, p\} \right\}.$$

Proposition 4.16 (well-definedness of the decade Unum lattice)**.** *Let $p \in \mathbb{N}_0$ and $s \in \mathbb{N}$. $P_D(p,s)$ is well-defined in terms of Definition 4.1.*

Proof. The desired property $|P_E(p,m)| = p$ follows from Definition 4.15. We show that

$$\forall i,j \in \{1,\dots,p\} : i > j : p_i > p_j.$$

This is trivial for $p = 1$. For $p > 1$ and $i \in \{1,\dots,p-1\}$ we note that for $m \in \mathbb{N}$ it holds that

$$\begin{aligned}
(i+1) \bmod m = 0 &\Rightarrow \left\{ \begin{array}{l} i \bmod m = m-1 \\ \exists n \in \mathbb{N}_0 : (i+1) = n \cdot m \end{array} \right\} \\
&\Rightarrow \left\{ \begin{array}{l} \left\lfloor \frac{i+1}{m} \right\rfloor = \lfloor n \rfloor = n \\ \left\lfloor \frac{i}{m} \right\rfloor = n-1 \end{array} \right\} \\
&\Rightarrow \left\lfloor \frac{i+1}{m} \right\rfloor = \left\lfloor \frac{i}{m} \right\rfloor + 1
\end{aligned}$$

and obtain

$$\begin{aligned}
p_{i+1} - p_i = {} & \left[1 + 10^{-(s-1)} \cdot \left((i+1) \bmod \left(10^s - 10^{s-1}\right)\right)\right] \cdot 10^{\left\lfloor \frac{i+1}{10^s - 10^{s-1}} \right\rfloor} - \\
& \left[1 + 10^{-(s-1)} \cdot \left(i \bmod \left(10^s - 10^{s-1}\right)\right)\right] \cdot 10^{\left\lfloor \frac{i}{10^s - 10^{s-1}} \right\rfloor} \\
\geq {} & \left[1 + 10^{-(s-1)} \cdot 0\right] \cdot 10^{\left\lfloor \frac{i}{10^s - 10^{s-1}} \right\rfloor + 1} - \\
& \left[1 + 10^{-(s-1)} \cdot \left(10^s - 10^{s-1} - 1\right)\right] \cdot 10^{\left\lfloor \frac{i}{10^s - 10^{s-1}} \right\rfloor} \\
= {} & \left[10 - 1 - 10^{-(s-1)+s} + 10^{-(s-1)+s-1} + 10^{-(s-1)}\right] \cdot 10^{\left\lfloor \frac{i}{10^s - 10^{s-1}} \right\rfloor} \\
= {} & 10^{-(s-1)} \cdot 10^{\left\lfloor \frac{i}{10^s - 10^{s-1}} \right\rfloor} \\
> {} & 0.
\end{aligned}$$

The proof is finished by showing that $p_i \in (1, \breve{\infty})$.

$$\begin{aligned}
p_i &= \left[1 + 10^{-(s-1)} \cdot \left(i \bmod (10^s - 10^{s-1})\right)\right] \cdot 10^{\left\lfloor \frac{i}{10^s - 10^{s-1}} \right\rfloor} \\
&\geq 1 + 10^{-(s-1)} \cdot \left(i \bmod (10^s - 10^{s-1})\right) \\
&> 1
\end{aligned}$$

□

Proposition 4.17 (maximum of the decade Unum lattice)**.** *Let $p \in \mathbb{N}_0$ and $s \in \mathbb{N}$. The maximum of the decade Unum lattice is*

$$\max\{P_D(p,s)\} = \left(1 + 10^{-(s-1)} \cdot \left[p \bmod \left(10^s - 10^{s-1}\right)\right]\right) \cdot 10^{\left\lfloor \frac{p}{10^s - 10^{s-1}} \right\rfloor}.$$

Proof. As shown in the proof of Proposition 4.16, $\forall i > j : p_i > p_j$ and thus

$$\max\{P_D(p,s)\} = p_p = \left(1 + 10^{-(s-1)} \cdot \left[p \bmod \left(10^s - 10^{s-1}\right)\right]\right) \cdot 10^{\left\lfloor \frac{p}{10^s - 10^{s-1}} \right\rfloor}. \qquad \square$$

Comparing the resulting distribution to an exponential curve fitted to the boundary-points, as shown in Figure 4.1, one can see that a nearly exponential distribution has been achieved. As we can see, the decade Unum lattice is a good compromise between a linear and an exponential Unum lattice.

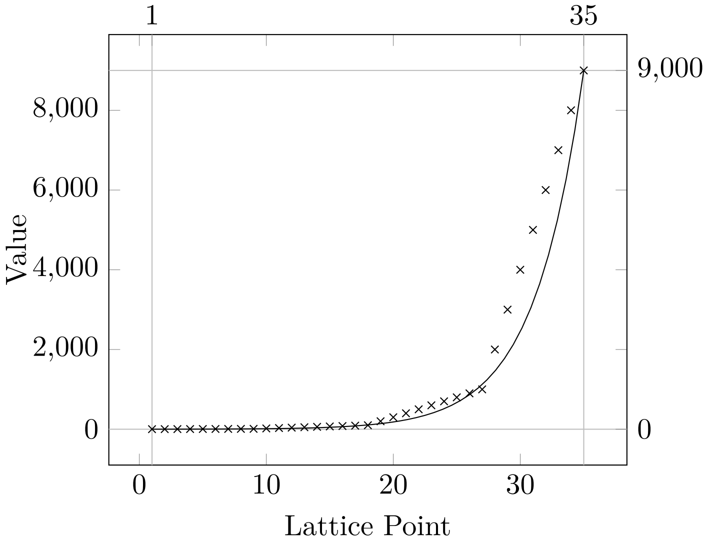

Figure 4.1.: $P_D(35,1)$ (demarked by crosses) in comparison with an exponential curve fitted to the endpoints $(0,0)$ and $(35, \max(P_D(35,1)))$.

4.2. Machine Implementation

The goal of a machine implementation for Unums is to find a model for $\mathcal{P}(\mathbb{U}(P))$ on a specially chosen lattice P. This means the ability to model subsets of $\mathbb{R}^*$ using multiple Unums, including degenerate intervals.

4.2.1. Unum Enumeration

We start off with the definition of the

Definition 4.18 (ascension operator)**.** *Let $(S,<)$ be a finite strictly ordered set. The ascension operator*

$$\text{asc}\colon S \times \{1,\ldots,|S|\} \to S$$

is defined for

$$s_i \in \{s_i \mid i \in \{1,\ldots,|S|\} \land s_1 < \ldots < s_{|S|}\} = S$$

as

$$(S, n) \mapsto s_n$$

Using the ascension operator, we enumerate the elements in $\mathbb{U}(P)$ with P as in Definition 4.1, taking note that $\mathbb{U}(P) \cap \mathcal{P}((0,1))$, $\mathbb{U}(P) \cap \mathcal{P}((1,\breve{\infty}))$, $\mathbb{U}(P) \cap \mathcal{P}((\breve{\infty},-1))$ and $\mathbb{U}(P) \cap \mathcal{P}((-1,0))$ are finite strictly ordered sets. In other words, we define a mapping from $\{0,\dots,|\mathbb{U}(P)|-1\}$, which is $\{0,\dots,8\cdot(|P|+1)-1\}$ according to Proposition 4.3, into $\mathbb{U}(P)$, called the

Definition 4.19 (Unum enumeration)**.** *Let P as in Definition 4.1. The Unum enumeration*

$$u\colon \{0,\dots,|\mathbb{U}(P)|-1\} \to \mathbb{U}(P)$$

is defined as

$$n \mapsto \begin{cases} \{0\} & n = 0\cdot(|P|+1) \\ \operatorname{asc}(\mathbb{U}(P)\cap\mathcal{P}((0,1)), n - 0\cdot(|P|+1)) & 0\cdot(|P|+1) < n < 2\cdot(|P|+1) \\ \{1\} & n = 2\cdot(|P|+1) \\ \operatorname{asc}(\mathbb{U}(P)\cap\mathcal{P}((1,\breve{\infty})), n - 2\cdot(|P|+1)) & 2\cdot(|P|+1) < n < 4\cdot(|P|+1) \\ \{\breve{\infty}\} & n = 4\cdot(|P|+1) \\ \operatorname{asc}(\mathbb{U}(P)\cap\mathcal{P}((\breve{\infty},-1)), n - 4\cdot(|P|+1)) & 4\cdot(|P|+1) < n < 6\cdot(|P|+1) \\ \{-1\} & n = 6\cdot(|P|+1) \\ \operatorname{asc}(\mathbb{U}(P)\cap(-1,0), n - 6\cdot(|P|+1)) & 6\cdot(|P|+1) < n < 8\cdot(|P|+1). \end{cases}$$

Remark 4.20 (enumeration of infinity)**.** *For arbitrary $\mathbb{U}(P)$ with P as in Definition 4.1 it follows that*

$$u\left(\frac{|\mathbb{U}(P)|}{2}\right) = \{\breve{\infty}\}.$$

To describe the enumeration intuïtively, we cut the $\mathbb{R}^*$-circle at 0 and trace all Unums from 0 to 0 in a counter-clockwise direction. In the machine the Unum enumeration mapping can be realised using unsigned integers. One can deduce that for a given number of *Unum bits* $n_b \in \mathbb{N}$ an unsigned n_b-bit integer can represent 2^{n_b} values, namely 0 through $2^{n_b}-1$.

Even though in theory the size of $|P|$ can be arbitrary, as it is the case for the provided toolbox, one must respect the fundamental data-types in a machine, resulting in the limitation $n_b \in \{8, 16, 32, 64, \dots\}$ in the interest of not wasting any bit patterns in the process. It follows that we are interested in finding out the required lattice size for a given n_b.

Proposition 4.21 (lattice size depending on Unum bits)**.** *Let $n_b \in \mathbb{N}$, $n_b > 2$ and P as in Definition 4.1. Given n_b Unum bits it follows that*

$$|P| = 2^{n_b-3} - 1.$$

Proof. With n_b Unum bits it follows that $|\mathbb{U}(P)| = 2^{n_b}$. According to Proposition 4.3 we know that $|\mathbb{U}(P)| = 8 \cdot (|P| + 1)$ and thus

$$2^{n_b} = 8 \cdot (|P| + 1) = 2^3 \cdot (|P| + 1) \quad \Leftrightarrow \quad |P| = 2^{n_b - 3} - 1 \qquad \square$$

According to the results obtained in Section 4.1, we will only take decade lattices into account. We are led to the

Definition 4.22 (set of machine Unums)**.** *Let $n_b \in \mathbb{N}$, $n_b > 2$ and $n_s \in \mathbb{N}$. The set of machine Unums with n_b bits and n_s significant digits is defined as*

$$\mathbb{U}_M(n_b, n_s) := \mathbb{U}(P_D(2^{n_b - 3} - 1, n_s).$$

Having found an expression for machine Unums, it is now possible to represent arbitrary elements of $\mathcal{P}(\mathbb{U}_M(n_b, n_s))$ in the machine to model sets of real numbers.

4.2.2. Operations on Sets of Real Numbers

Unums alone are not very useful for arithmetic purposes, given the nature of dual Unum operations (see Definition 4.7), which we want to illustrate with the following example.

Example 4.23. *Let $P = \{2, 3.5, 5, 6\}$, which satisfies Definition 4.1. We see that $(1, 2), \{3.5\} \in \mathbb{U}(P)$, but*

$$\mathrm{bl}\left((1, 2) \boxplus \{3.5\}\right) = \mathrm{bl}\left((4.5, 5.5)\right) = \{(3.5, 5), \{5\}, (5, 6)\} \notin \mathbb{U}(P).$$

The basic datatype, thus, has to be an element of $\mathcal{P}(\mathbb{U}_M(n_b, n_s))$. Given this set is finite with $2^{(|\mathbb{U}_M(n_b, n_s)|)} = 2^{2^{n_b}}$ elements, a bit string of length 2^{n_b} can represent all elements of $\mathcal{P}(\mathbb{U}_M(n_b, n_s))$. We call this bit string a ‘SORN’ for ‘set of real numbers’.

Operations on SORNs are carried out in the machine by having lookup tables (LUTs) for $\mathrm{bl}(u(i) \star u(j))$, where $\star \in \{\boxplus, \boxtimes\}$ is an $\mathbb{R}^*$-Flake-operation evaluated for arbitrary Unum-indices $i, j \in \{0, \dots, 2^{n_b} - 1\}$. Given $\boxplus$ and $\boxtimes$ are associative, limiting the lookup table to $i \leq j$ is sufficient, resulting in a triangular array for each operation.

The results $\mathrm{bl}(u(i) \star u(j))$, being connected subsets of $\mathbb{U}_M(n_b, n_s)$, can be expressed as an oriented range $[u(m), u(n)]$ with $m, n \in \{0, \dots, 2^{n_b} - 1\}$ and $m \leq n$, containing all Unums between $u(m)$ and $u(n)$. This can be stored in the machine as indices $\{m, n\}$ each taking up n_b bit of storage. Thus, each table entry takes up $2 \cdot n_b$ bit of storage.

Proposition 4.24 (size of LUTs)**.** *The Unum LUTs for $\boxplus$ and $\boxtimes$ take up $n_b \cdot 2^{n_b + 1} \cdot (2^{n_b} + 1)$ bit.*

Proof. With 2^{n_b} rows, we know that each LUT has $\sum_{i=1}^{2^{n_b}} i$ entries. Using the Gauß summation formula and the facts that each entry takes up $2 \cdot n_b$ bit and we have two operations and, thus, two LUTs, the total storage size is

$$2 \cdot (2 \cdot n_b) \cdot \left(\sum_{i=1}^{2^{n_b}} i \right) \text{bit} = 4 \cdot n_b \cdot \left(\frac{2^{n_b} \cdot (2^{n_b} + 1)}{2} \right) \text{bit} = n_b \cdot 2^{n_b + 1} \cdot (2^{n_b} + 1) \,\text{bit}. \qquad \square$$

With the lookup tables constructed, operations on SORNs are analogous to dual Unum operations (see Definition 4.7), with the only difference that the set union for the bit strings is realised with a bitwise OR.

4.2.3. Unum Toolbox

To examine the numerical properties of Unums, there needs to be a toolbox to see how this concept works out inside the machine. The reason why a new toolbox was developed in the course of this thesis is that all other toolboxes available at the time of writing are not using LUTs to do calculations. Instead, they emulate Unum-arithmetic with floating-point numbers that are mapped to a given lattice.

To give an answer to the question if Unums could in theory replace floating-point numbers for some applications, it is necessary to avoid floating-point arithmetic at runtime as much as possible. A possible future machine implementing Unums in hardware would also be constrained to LUTs and would not be able to use floating-point numbers in the process and at the same time leverage the energy and complexity savings projected by GUSTAFSON in [Gus16b].

The Unum toolbox programmed in the course of this thesis and used to examine the numerical behaviour of Unums in Section 4.3 is split up in two parts. The first part is the environment generator `gen` (see Listing B.2.1), generating the LUTs in `table.c`, based on type definitions in `table.h` (see Listing B.2.2) and the environment parametres in `config.mk` (see Listing B.2.4), and the lattice-specific toolbox-header `unum.h`. The choice of lattice points can be arbitrary and it is relatively simple to extend the generator, but because of the results obtained in Section 4.1 only the generating function for a decade Unum lattice is implemented (see `gendeclattice()` in Listing B.2.1).

The second part is the toolbox itself (see Listing B.2.3), working with the previously generated `table.c` and `unum.h`, but being lattice-agnostic in general. The fundamental data type for operations is `SORN` defined in `unum.h`, corresponding to the SORN-concept constructed earlier. Just as proposed by GUSTAFSON in [Gus16b, Section 3.2], the SORN is a bit array on which operations are carried out as proposed and close to how it would happen in a native machine implementation.

The provided toolbox functions (see `unum.h` and Listing B.2.3) are of both arithmetic and set theoretical nature. The arithmetic functions corresponding to addition and subtraction are `uadd()` and `usub()`. Addition in this context means the dual Unum operation $\langle\boxplus\rangle$ using the addition LUT `addtable` in `table.c`. Subtraction is achieved by negating the second argument on a per-Unum basis and performing an addition, preserving set-dependencies if present. Analogously, there are `umul()` and `udiv()` for multiplication and division using $\langle\boxtimes\rangle$ and the multiplication LUT `multable` in `table.c`. The arithmetic functions `uneg()` and `uinv()` negate and invert a SORN respectively on a per-Unum basis corresponding to the $\mathbb{R}^*$-Flake negation '$-$' and inversion '/'. The function `uabs()` corresponds to a Unum modulus function and the `ulog()` function is an implementation of the ln function on Unums using the LUT `logtable`.

SORN operations and modifications are generalised in the functions `_sornop()` and `_sornmod()` in Listing B.2.3 respectively. They are the foundation for almost all arithmetic functions of this toolbox. Dependent sets are detected by comparing the two pointers to the operands passed to the arithmetic functions. If they are equal, the sets are dependent.

The set theoretical functions are `uemp()` and `uset()` for emptying and setting SORNs, `ucut()` and `uuni()` for cutting and taking the union of two SORNs and `uequ()` and `usup()` to check if two SORNs are equal and if one SORN is the superset of another.

The input and output functions play a special role in this toolbox. `uint()` is the only function using floating point numbers to add a closed interval to a SORN and `uout()` prints a SORN in a human-readable format to standard output.

When using the Unum toolbox, only the components `unum.h` and the static library `libunum.a` are relevant and need to be present when compiling programs using the Unum toolbox (see Section B.3). All functions are reëntrant and, thus, thread-safe.

4.3. Revisiting Floating-Point-Problems

Using the toolbox presented in Subsection 4.2.3, we implement the IEEE 754 floating-point problems studied in Section 2.4 and examine their behaviour within the Unum arithmetic. For all examples in this section the environment was set to $(n_b, n_s) = (12, 2)$.

4.3.1. The Silent Spike

We can express the spike function (2.2) within the Unum arithmetic, using a LUT-based natural logarithm

$$\mathrm{LN}\colon \mathcal{P}(\mathbb{U}(P)) \to \mathcal{P}(\mathbb{U}(P))$$

defined as

$$U \mapsto \begin{cases} \langle \ln_{\mathbb{F}} \rangle(U) & U \cap \mathcal{P}((\breve{\infty}, 0]) = \emptyset \\ \emptyset & \text{else} \end{cases}$$

(see `ulog()` in Listing B.2.3) and an elementary Unum modulus function $|\cdot|$ (see `uabs()` in Listing B.2.3), as

$$F(X) := \mathrm{LN}(|\,\mathrm{bl}(\{3\}) \boxtimes (\mathrm{bl}(\{1\}) \boxplus -X) \boxplus \mathrm{bl}(\{1\})|). \tag{4.1}$$

As we have previously evaluated f in an environment of all floating-point numbers of the singularity at $\frac{4}{3}$ (see Figure 2.2), we evaluate F in an environment of all Unums of the singularity $\mathrm{bl}\left(\frac{4}{3}\right)$ using the Unum toolbox (see Listing B.3.4). The behaviour is exhibited in Figure 4.2 and it can be observed that the spike is not hidden any more as was the case with the floating-point implementation.

This shows that Unums can effectively be used to quickly evaluate guaranteed bounds for a given function and observe singular behaviour without taking the risk of missing it. The bounds are guaranteed as the foundation for the Unum arithmetic are the well-defined operations on $\mathbb{R}^*$-Flakes (see Definition 3.13 and Theorem 3.14).

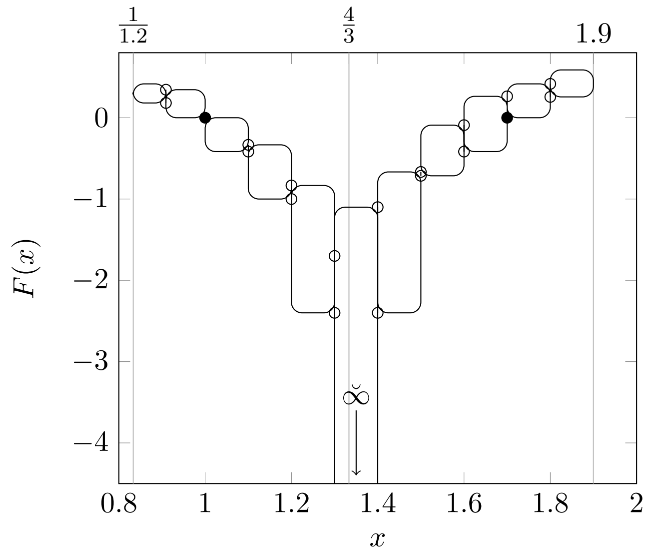

Figure 4.2.: Evaluation of the Unum spike function F (see (4.1)) on all Unums in $[\frac{1}{1.2}, 1.9]$ with $(n_b, n_s) = (12, 2)$ ($\circ/\bullet$ demarks open/closed interval endpoints); see Listing B.3.4.

4.3.2. Devil's Sequence

The devil's sequence is translated into Unum arithmetic by transforming (2.3) into the equivalent SORN-sequence

$$U_n := \begin{cases} \mathrm{bl}(\{2\}) & n = 0 \\ \mathrm{bl}(\{-4\}) & n = 1 \\ \mathrm{bl}(\{111\}) \boxplus - \mathrm{bl}(\{1130\}) \boxtimes /U_{n-1} \boxplus \mathrm{bl}(\{3000\}) \boxtimes /(U_{n-1} \boxtimes U_{n-2}) & n \geq 2. \end{cases}$$

Running the Unum toolbox implementation (see Listing B.3.2) of this problem, we obtain

$$U_{25} = \mathbb{R}^*.$$

This indicates the instability of the problem posed. Even though the information loss is great, this result can at least be a warning to investigate the numerical behaviour of the given sequence.

4.3.3. The Chaotic Bank Society

Taking a look at the chaotic bank society problem, we determine the equivalent SORN-sequence to (2.5) as

$$A_n := \begin{cases} A_0 & n = 0 \\ A_{n-1} \boxtimes \mathrm{bl}(\{n\}) \boxplus - \mathrm{bl}(\{1\}) & n \geq 1. \end{cases}$$

Again, running the Unum toolbox implementation (see Listing B.3.3), we obtain for $A_0 = \mathrm{bl}(\{e-1\}) = (1.7, 1.8)$

$$A_{25} = \mathbb{R}^*.$$

This is consistent with the theoretical results we obtained, given we can find an $\varepsilon > 0$ such that A_0 contains $e - 1 + \delta$ with $\delta \in (-\varepsilon, \varepsilon)$, as $e - 1 \notin P_D(2^{n_b-3} - 1, n_d)$.
We observe that, even though the results do not lie about the solution, the information loss is great.

Concluding, introducing Unums as a number format allowing you to neglect stability analysis has turned out to be a false promise. We can also not sustain the notion that naïvely implementing algorithms in Unums abolishes the need for a break condition. Besides complete information loss, sticking- and creeping-effects elaborated in Subsection 4.4.2 additionally make it difficult to think of proper ways to do that.

4.4. Discussion

With the theoretical formulation of Unums and practical results, it is now time to discuss the format taking into account the results obtained in the previous chapters.

4.4.1. Comparison to IEEE 754 Floating-Point Numbers

It is of central interest to see how the Unums hold up to the previously introduced IEEE 754 floating-point numbers. To illustrate the behaviour of the machine Unums, different parametres of the systems are laid out in Table 4.1.

n_b (bit)	8	16	32	64
n_s	1	3	7	15
$\lvert P_D \rvert$	$= 3.10 \cdot 10^{+1}$	$\approx 8.19 \cdot 10^{+3}$	$\approx 5.37 \cdot 10^{+8}$	$\approx 2.31 \cdot 10^{+18}$
$\lvert \mathbb{U}_M \rvert$	$= 2.56 \cdot 10^{+2}$	$\approx 6.55 \cdot 10^{+4}$	$\approx 4.29 \cdot 10^{+9}$	$\approx 1.84 \cdot 10^{+19}$
$\max(P_D)$	$= 5.00 \cdot 10^{+3}$	$= 1.91 \cdot 10^{+9}$	$\approx 6.87 \cdot 10^{+59}$	$\approx 1.43 \cdot 10^{+2562}$
$\max(P_D)^{-1}$	$= 2.00 \cdot 10^{-4}$	$\approx 5.24 \cdot 10^{-10}$	$\approx 1.45 \cdot 10^{-60}$	$\approx 6.99 \cdot 10^{-2563}$
Size of LUTs	$\approx$ 132 kB	$\approx$ 17 GB	$\approx 1.48 \cdot 10^{20}$ B	$\approx 5.44 \cdot 10^{39}$ B

Table 4.1.: Machine Unums properties for $n_b \in \{8, 16, 32, 64\}$ and n_s selected to match IEEE 754 significant decimal digits ($= \lfloor \log_{10}(2^{n_m+1}) \rfloor$) for each storage size.

Comparing Table 4.1 to Table 2.1, we note that for the same number of storage bits, the dynamic range, the ratio of the largest and smallest representable numbers, of Unums is orders of magnitude larger than that of IEEE 754 floating-point numbers. For example, with a storage size of 16 bit, the dynamic range of IEEE 754 floating-point numbers is

$$\frac{\max(\mathbb{M}_1)}{\min(\mathbb{M}_0 \cap \mathbb{R}^+_{\neq 0})} \approx \frac{6.55 \cdot 10^{+4}}{5.96 \cdot 10^{-8}} \approx 1.10 \cdot 10^{12}.$$

For Unums, we obtain

$$\frac{\max(P_D)}{\max(P_D)^{-1}} \approx \frac{1.91 \cdot 10^{+9}}{5.24 \cdot 10^{-10}} \approx 3.65 \cdot 10^{18}$$

respectively, which is an increase of roughly 6 orders of magnitude. The reason for this significant difference is the fact that no bit patterns are wasted for NaN-representations in the Unum number format.

One the other hand, one can see that any values for n_b beyond roughly 12 bit (corresponding to a LUT size of $\approx$ 50 MB) is not feasible given the huge size of the LUTs. It shows that we can only really reason about machine Unum environments with $n_b \in \{3, \dots, 12\}$.

4.4.2. Sticking and Creeping

Working with the Unum toolbox, two effects seem to influence iterative calculations substantially. A fitting description would be to call them *sticking-* and *creeping-effects* respectively. They can be observed, for instance, when evaluating infinite series within the Unum arithmetic, and this example will be examined here.

Example 4.25 (Euler's number). *Determining* Euler*'s number in the Unum arithmetic can be done by defining a SORN-series E_n satisfying*

$$\mathrm{bl}(\{e\}) \in \lim_{n \to \infty} (E_n),$$

where

$$E_n := \overset{n}{\underset{k=0}{\langle \boxplus \rangle}} \left[/ \overset{k}{\underset{\ell=0}{\langle \boxtimes \rangle}} \mathrm{bl}(\{\ell\}) \right], \tag{4.2}$$

which corresponds to the partial sums of the infinite series representation of e as

$$e = \sum_{k=0}^{\infty} \frac{1}{k!},$$

Using the Unum toolbox (see Listing B.3.1), the partial sums of this problem are visualised in Figure 4.3. The first 21 *iterates are depicted and illustrate a pathological behaviour.*

Starting from $n = 3$, the lower bound of the solution set is stuck at the value 2.6. One can also observe that the upper bound is growing linearly on each iteration. It creeps away from e and reduces the quality of the solution with each step.

The cause of these sticking- and creeping-effects is the fact that we add infinitesimally small values to the SORN on each iteration. The lower bound gets stuck because the value added is smaller than the length of the lowest interval, hitting a blind spot of the blur function. The upper bound creeps away because even though we add an infinitesimally small value, it expands to at least the next following Unum value.

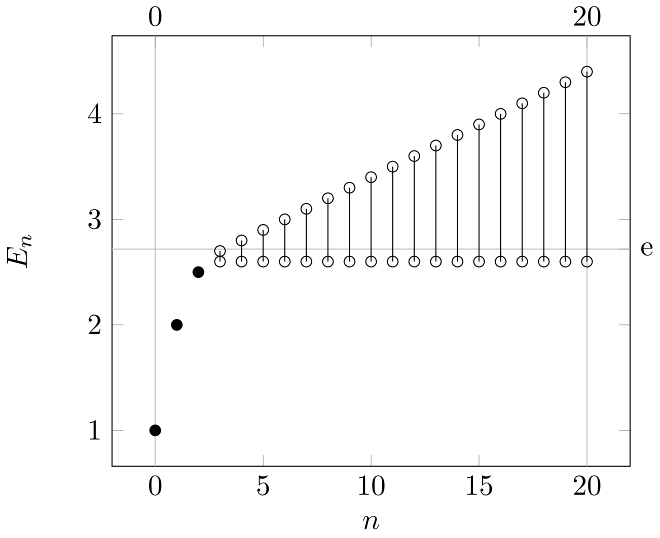

Figure 4.3.: Evaluation of the Unum Euler partial sums (4.2) for iterations $n \in \{0, \ldots, 20\}$ with $(n_b, n_s) = (12, 2)$ ($\circ$/$\bullet$ demarks open/closed interval endpoints); see Listing B.3.1.

This problem makes it impossible to work with Unums to examine infinite series or sequences and iterative problems in general. Even though Unums do not lie about the solution, the quality of it is decreased on each iteration, as we could already see in Subsections 4.3.2 and 4.3.3. There is also no chance of formulating a break condition for the given algorithm because of this behaviour. We observe comparable problems for finding break conditions for infinite series that do not converge quickly using floating-point numbers, so we can generally think of it as an unsolved problem following from the finite nature of the machine.

4.4.3. Lattice Switching

A strong theoretical advantage is that one could evaluate an expression on a set of Unums with a coarse lattice first and then refine the lattice as soon as the set of possible solutions shrinks. It is questionable how this could be possible within the machine. One may find ways to reduce the size of the lookup tables, but assuming multiple different lattice-precisions including all LUTs would take up massive amounts of space on a microchip. Additionally, there needs to be a theory on how existing SORNs are translated between the different lattices, which might require its own set of LUTs for each transition, greatly increasing complexity.

If the solution space is only observed within small bounds, considerable amounts of memory are wasted for set representations beyond the bounds using a naïve SORN representation. The SORNs may be needed for intermediate values of a calculation, which could easily expand beyond the bounds of the solution space, but not for the final results.

This problem can be approached using a run-length encoding for SORNs comparable to how LUTs were implemented (see Subsection 4.2.2), but this would make SORN operations in general less efficient unless the operations take place directly on top of Unum enumeration indices.

4.4.4. Complexity

Despite the efforts to simplify arithmetic operations and overhead by creating lookup tables and working on bit strings in a simple manner, the cost of this simplification weighs heavily. The contradiction lies within the fact that to at least reduce the detrimental effects of sticking and creeping it is necessary to increase the number of Unum bits n_b. However, this is only possible up to a certain point until the LUTs become too large. In this context, dealing with strictly monotonic functions like ln in the Unum context requires LUTs for each of them as well (see Subsection 4.3.1).

It is questionable how useful the Unum arithmetic is within the tight bounds set by these limiting factors. However, it should be taken into account that there are possible uses for Unums on very coarse grids, for instance inverse kinematics. GUSTAFSON also identifies the problem (see [Gus16b, Section 6]) and notes that this problem could indicate that Unums are ‘ [...] primarily practical for low-accuracy but high-validity applications, thereby complementing float arithmetic instead of replacing it. ’ [Gus16b, Section 6.2]

5. Summary and Outlook

In the course of this thesis we started off with the construction of a mathematical description of IEEE 754 floating-point numbers, compared the properties of different binary storage formats and studied examples which uncover inherent weaknesses of this arithmetic.

Following from these observations, we constructed the projectively extended real numbers based on a small set of axioms. After introducing a definition of finite and infinite limits on the projectively extended real numbers, we showed their well-definedness in terms of these limits. Based on this foundation, we developed the Flake arithmetic and proved well-definedness in terms of set theory.

This effort led us to the mathematical foundation of Unums, which as proposed approaches the interval arithmetic dependency problem in a new way and is meant to be easy to implement in the machine. We presented different types of Unum lattices, evaluated the requirements for hardware implementations and studied the numerical behaviour in a Unum toolbox, which was developed in the course of this thesis. Using these results, we were able to draw the conclusion that Unums may not be a number format allowing naïve computations, but exhibited promising results in low-precision but high-validity applications.

The author expected to find drawbacks of this nature for the Unum number format, as any numerical system exhibits its strength only within certain conditions, making it easy to find examples where it fails. In this context, it was observed that IEEE 754 floating-point numbers and Unums complement each other. Given the nature of Unum arithmetic, it may be on the one hand difficult to do stability analysis due to the complexity of the arithmetic rules, but on the other hand the guaranteed bounds of the result do not cover up when an algorithm is not fit for this environment and indicate the need to approach the problem using a different numerical approach.

At the point of writing, the revised Unum format was approached with neither a mathematical foundation nor formalisation. The available toolboxes were only emulating Unums using floating-point arithmetic, hiding numerous drawbacks with regard to the complexity of lookup tables. The results obtained in this thesis make it possible to reason about Unums in the bounds that will also be present when it comes to implementing Unums in hardware and not only in software.

In general it is questionable if the approach of using lookup tables is really the best way to go, despite the possible advantage of simplifying calculations. It is questionable if it is really worth it to throw the entire IEEE 754 floating-point infrastructure overboard and have two exclusive numerical systems.

The bl operator presented in this thesis corresponds to the rounding operation for floating-point numbers to a certain extent. A topic for further research could be to introduce closed $\mathbb{R}^*$-intervals for $a, b \in \mathbb{R}^*$ with

$$[a, b] := \{a\} \sqcup (a, b) \sqcup \{b\} \subset \mathbb{F}.$$

Operations on $\mathbb{R}^*$-Flakes were shown to be well-defined and, thus, it is possible to extend Flake operations $\star \in \{\boxplus, \boxtimes\}$ to automatically well-defined operations $\diamond$ on closed intervals with

$$\begin{aligned}[a, b] \diamond [c, d] := {} & \{a\} \star \{c\} \cup \{a\} \star (c, d) \cup \{a\} \star \{d\} \cup \\ & (a, b) \star \{c\} \cup (a, b) \star (c, d) \cup (a, b) \star \{d\} \cup \\ & \{b\} \star \{c\} \cup \{b\} \star (c, d) \cup \{b\} \star \{d\}\end{aligned}$$

and simplify it accordingly. Discretisation is achieved by using floating-point numbers for the interval bounds and directed rounding for guaranteed bounds, as elaborated in (2.1).

Unums present the need to have lookup tables for every operation and nearly every elementary function to be feasible, which is a huge complexity problem. This is the reason why using floats instead of lookup tables to achieve this makes sense, because we studied the behaviour of strictly monotonic functions on Flakes in this thesis and can directly use strictly monotonic floating-point functions for Flake arithmetic instead of lookup tables. In the end, this could combine the accuracy of floating point numbers and the certainty of interval arithmetic. The difference between this and ordinary interval arithmetic using floating-point number bounds (see [IEE15]) is the use of the projectively extended real numbers instead of the affinely extended real numbers, making it possible to model degenerate intervals and divide by zero, and the knowledge of the results obtained in this thesis to approach the dependency problem. It comes at the cost of a total order relation and only offers a partial order, which can be assumed to be a smaller problem than it seems.

In the end, it all boils down to the question if using two 32 bit IEEE 754 floating-point numbers to model such a closed interval is better than using a single 64 bit IEEE 754 floating-point number for a diverse set of algorithms. The strategy of finding a solution by going from a coarse to a fine grid for Unums could be easily realised with floating-point bounded closed intervals and also prove to be useful for certain applications.

Reaching a point where the dynamic range of high-bit floating-point numbers is exceeding the range of numbers of usable magnitude only to compensate rounding errors to a certain extent, we might find interval arithmetic on the projectively extended real numbers to be a good future direction for improving the results of the very calculations we are doing every day.

A. Notation Directory

A.1. Section 2: IEEE 754 Floating-Point Arithmetic

n_m	number of mantissa-digits ($\equiv$ mantissa-bits in base-2)
n_e	number of exponent-bits
$\mathbb{M}_0(n_m, \underline{e})$	set of subnormal floating-point numbers; see Definition 2.2
$\mathbb{M}_1(n_m, \underline{e}, \overline{e})$	set of normal floating-point numbers; see Definition 2.1
$\mathbb{M}(n_m, \underline{e} - 1, \overline{e} + 1)$	set of floating-point numbers; see Definition 2.3
$\lvert \mathrm{NaN} \rvert (n_m)$	number of NaN representations; see Proposition 2.7
$\underline{e}(n_e)$, $\overline{e}(n_e)$	exponent bias; see Definition 2.8
$\mathrm{rd}_{\mathcal{E}}$	nearest and tie to even rounding; see Definition 2.12
$\mathrm{rd}_{\uparrow}$	upward rounding; see Definition 2.13
$\mathrm{rd}_{\downarrow}$	downward rounding; see Definition 2.14

A.2. Section 3: Interval Arithmetic

$\mathbb{R}^*$	projectively extended real numbers; see Definition 3.1
$\breve{\infty}$	infinity symbol of $\mathbb{R}^*$; see Definition 3.1
$\sqcup$	disjoint union; see Definition 3.7
$(\underline{a}, \overline{a})$	open $\mathbb{R}^*$-interval between $\underline{a}$ and $\overline{a}$; see Definition 3.8
$\mathbb{I}$	set of open $\mathbb{R}^*$-intervals; see Definition 3.9
$\oplus$	addition operator on $\mathbb{I}$; see Definition 3.9
$\otimes$	multiplication operator on $\mathbb{I}$; see Definition 3.9
$\S(S)$	set of S-singletons; see Definition 3.12
$\mathbb{F}$	set of $\mathbb{R}^*$-Flakes; see Definition 3.13
$\boxplus$	addition operator on $\mathbb{F}$; see Definition 3.13
$\boxtimes$	multiplication operator on $\mathbb{F}$; see Definition 3.13
$f_{\mathbb{F}}$	$\mathbb{R}^*$-Flake evaluation of the strictly monotonic function f; see Definitions 3.15 and 3.17

A.3. Section 4: Unum Arithmetic

$\mathbb{U}(P)$	set of Unums on the lattice P; see Definition 4.1
$\mathcal{P}(S)$	powerset of S; see Definition 4.4
bl	blur operator; see Definition 4.5
$\langle\star\rangle$	dual Unum operation; see Definition 4.7
$\langle f_{\mathbb{F}}\rangle$	Unum evaluation of the strictly monotonic function f; see Definitions 4.9 and 4.10
$P_L(p,n)$	linear Unum lattice; see Definition 4.11
$P_E(p,m)$	exponential Unum lattice; see Definition 4.13
$P_D(p,s)$	decade Unum lattice; see Definition 4.15
asc	ascension operator; see Definition 4.18
$u(n)$	nth Unum in the Unum enumeration; see Definition 4.19
n_b	number of Unum bits
n_d	number of significant digits
$\mathbb{U}_M(n_b,n_s)$	set of machine Unums; see Definition 4.22

B. Code Listings

B.1. IEEE 754 Floating-Point Problems

B.1.1. spike.c

```
#include <float.h>
#include <math.h>
#include <stdio.h>

#define LEN(x) (sizeof (x) / sizeof *(x))
#define NUMPOINTS (10)
#define POLE (4.0/3.0)

int
main(void)
{
        double x[NUMPOINTS * 2 + 1][2];
        int i;

        /* fill POINT environment */
        x[NUMPOINTS][0] = POLE;
        for (i = NUMPOINTS - 1; i >= 0; i--) {
                x[i][0] = nextafter(x[i + 1][0], -INFINITY);
        }
        for (i = NUMPOINTS + 1; i < NUMPOINTS * 2 + 1; i++) {
                x[i][0] = nextafter(x[i - 1][0], +INFINITY);
        }

        /* calculate values of |3*(1-x)+1| in the POINT environment */
        for (i = 0; i < NUMPOINTS * 2 + 1; i++) {
                x[i][1] = fabs(3 * (1 - x[i][0]) + 1);
                printf("x-%.2f=%.20e␣|3*(1-x)+1|=%.20f\n", POLE,
                       x[i][0]-POLE, x[i][1]);
        }

        putchar('\n');

        /* calculate values of f(x) in the POINT environment */
```

```
        for (i = 0; i < NUMPOINTS * 2 + 1; i++) {
                x[i][1] = log(fabs(3 * (1 - x[i][0]) + 1));
                printf("x-%.2f=%.20e␣f(x)=%.20f\n", POLE,
                        x[i][0]-POLE, x[i][1]);
        }

        return 0;
}
```

B.1.2. devil.c

```
#include <stdio.h>

int
main(void)
{
        double a, b, tmp;
        int i;

        a = 2;
        b = -4;

        for (i = 2; i < 26; i++) {
                tmp = 111 - 1130 / b + 3000 / (b * a);
                a = b;
                b = tmp;
                printf("u_%.2d␣=␣%f\n", i, b);
        }

        return 0;
}
```

B.1.3. bank.c

```
#include <float.h>
#include <math.h>
#include <stdio.h>

const int years = 25;

int
main(void)
```

```
{
        double a;
        int n;

        a = 1.718281828459045235;

        for (n = 1; n <= years; n++) {
                a = a * n - 1;
        }

        printf("u_%d␣=␣%f\n", years, a);

        return 0;
}
```

B.1.4. Makefile

```
PROBLEMS = devil bank
LMPROBLEMS = spike

all: $(PROBLEMS) $(LMPROBLEMS)

$(LMPROBLEMS): LDFLAGS = -lm

%: %.c
        cc $^ -o $@ $(LDFLAGS)
clean:
        rm -f $(PROBLEMS) $(LMPROBLEMS)
```

B.2. Unum Toolbox

B.2.1. gen.c

```
#include <fenv.h>
#include <float.h>
#include <math.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#undef LEN
```

```
#define LEN(x) sizeof(x) / sizeof(*x)
#undef UCLAMP
#define UCLAMP(i, off) (((((off < 0) && (i) < -off) ? \
                       numunums - ((-off - (i)) % numunums) : \
                       ((off > 0) && (i) + off > numunums - 1) ? \
                       ((i) + off % numunums) % numunums : (i) + off)) \
                       % numunums)
#undef MIN
#define MIN(x,y)  ((x) < (y) ? (x) : (y))
#undef MAX
#define MAX(x,y)  ((x) > (y) ? (x) : (y))

struct _unum {
        double val;
        char *name;
};

struct _unumrange {
        size_t low;
        size_t upp;
};

struct _latticep {
        char *name;
        double val;
};

static void
printunums(struct _unum *unum, size_t numunums)
{
        size_t i;

        fputs("\nstruct␣_unum␣unums[]␣=␣{\n", stdout);

        for (i = 0; i < numunums; i++) {
                if (isnan(unum[i].val) && !unum[i].name) {
                        printf("\t{␣NAN,␣NULL␣},\n");
                } else if (isinf(unum[i].val)) {
                        printf("\t{␣INFINITY,␣\"%s\"␣},\n",
                               unum[i].name);
                } else {
                        printf("\t{␣%f,␣\"%s\"␣},\n", unum[i].val,
                               unum[i].name);
                }
```

```
	}

	fputs("};\n", stdout);
}

size_t
blur(double val, struct _unum *unum, size_t numunums)
{
	size_t i;

	/* infinity is infinity */
	if (isinf(val)) {
		return numunums / 2;
	}

	for (i = 0; i < numunums; i++) {
		/* equality within relative epsilon */
		if (isfinite(unum[i].val) && fabs(unum[i].val - val) <=
		    DBL_EPSILON * MAX(fabs(unum[i].val), fabs(val))) {
			return i;
		}

		/* in range */
		if (isnan(unum[i].val) &&
		    val < unum[UCLAMP(i, +1)].val &&
		    val > unum[UCLAMP(i, -1)].val) {
			return i;
		}
	}

	/* negative or positive range outshot */
	return (val < 0) ? (numunums / 2 + 1) : (val > 0) ?
	       (numunums / 2 - 1) : 0;
}

void
add(size_t a, size_t b, struct _unumrange *res,
    struct _unum *unum, size_t numunums)
{
	double av, bv, aupp, alow, bupp, blow;

	av = unum[a].val;
	bv = unum[b].val;

```

```
        if (isnan(av) && isnan(bv)) {
                /*
                 * a interval, b interval
                 */
                aupp = unum[UCLAMP(a, +1)].val;
                alow = unum[UCLAMP(a, -1)].val;
                bupp = unum[UCLAMP(b, +1)].val;
                blow = unum[UCLAMP(b, -1)].val;

                if ((isinf(alow) && isinf(aupp)) ||
                    (isinf(blow) && isinf(bupp))) {
                        /* all real numbers */
                        res->low = numunums / 2 + 1;
                        res->upp = numunums / 2 - 1;
                        return;
                } else if (isinf(alow) && isinf(blow)) {
                        /* (iffy, aupp + bupp) */
                        res->low = numunums / 2 + 1;
                        fesetround(FE_UPWARD);
                        res->upp = blur(aupp + bupp, unum,
                                        numunums);
                } else if (isinf(aupp) && isinf(bupp)) {
                        /* (alow + blow, iffy) */
                        fesetround(FE_DOWNWARD);
                        res->low = blur(alow + blow, unum, numunums);
                        res->upp = numunums / 2 + 1;
                } else {
                        /* (alow + blow, aupp + bupp) */
                        fesetround(FE_DOWNWARD);
                        res->low = blur(alow + blow, unum, numunums);
                        fesetround(FE_UPWARD);
                        res->upp = blur(aupp + bupp, unum, numunums);
                }
        } else if (!isnan(av) && !isnan(bv)) {
                /*
                 * a point, b point
                 */
                if (isinf(av) && isinf(bv)) {
                        /* all extended real numbers */
                        res->low = 0;
                        res->upp = numunums - 1;
                        return;
                } else {
                        fesetround(FE_DOWNWARD);
```

```
                        res->low = blur(av + bv, unum, numunums);
                        fesetround(FE_UPWARD);
                        res->upp = blur(av + bv, unum, numunums);
                }
        } else if (!isnan(av) && isnan(bv)) {
                /*
                 * a point, b interval
                 */
                bupp = unum[UCLAMP(b, +1)].val;
                blow = unum[UCLAMP(b, -1)].val;

                if (isinf(av)) {
                        /* all extended real numbers */
                        res->low = 0;
                        res->upp = numunums - 1;
                        return;
                } else {
                        fesetround(FE_DOWNWARD);
                        res->low = blur(av + blow, unum, numunums);
                        fesetround(FE_UPWARD);
                        res->upp = blur(av + bupp, unum, numunums);
                }
        } else if (!isnan(bv) && isnan(av)) {
                /*
                 * a interval, b point
                 */
                add(b, a, res, unum, numunums);
                return;
        }

        if (isnan(av) || isnan(bv)) {
                /* we had an open interval in our calculation
                 * and need to check if res->upp or res->low
                 * are a point. If this is the case, we have
                 * to round it down to respect the openness
                 * of the real interval */
                if (!isnan(unum[res->low].val)) {
                        res->low = UCLAMP(res->low, +1);
                }
                if (!isnan(unum[res->upp].val)) {
                        res->upp = UCLAMP(res->upp, -1);
                }
        }
}
```

```

void
mul(size_t a, size_t b, struct _unumrange *res,
    struct _unum *unum, size_t numunums)
{
        double av, bv, aupp, alow, bupp, blow;

        av = unum[a].val;
        bv = unum[b].val;

        if (isnan(av) && isnan(bv)) {
                /*
                 * a interval, b interval
                 */
                aupp = unum[UCLAMP(a, +1)].val;
                alow = unum[UCLAMP(a, -1)].val;
                bupp = unum[UCLAMP(b, +1)].val;
                blow = unum[UCLAMP(b, -1)].val;

                if ((isinf(alow) && isinf(aupp)) ||
                    (isinf(blow) && isinf(bupp))) {
                        /* all real numbers */
                        res->low = numunums / 2 + 1;
                        res->upp = numunums / 2 - 1;
                        return;
                } else if (isinf(alow) && isinf(blow)) {
                        if (aupp <= 0 && bupp <= 0) {
                                /* (aupp * bupp, iffy) */
                                fesetround(FE_DOWNWARD);
                                res->low = blur(aupp * bupp,
                                                unum, numunums);
                                res->upp = numunums / 2 - 1;
                        } else {
                                /* all real numbers */
                                res->low = numunums / 2 + 1;
                                res->upp = numunums / 2 - 1;
                                return;
                        }
                } else if (isinf(aupp) && isinf(bupp)) {
                        if (alow >= 0 && blow >= 0) {
                                /* (alow * blow, iffy) */
                                fesetround(FE_DOWNWARD);
                                res->low = blur(alow * blow,
                                                unum, numunums);
```

```
                                res->upp = numunums / 2 - 1;
                        } else {
                                /* all real numbers */
                                res->low = numunums / 2 + 1;
                                res->upp = numunums / 2 - 1;
                                return;
                        }
                } else if (isinf(alow) && isinf(bupp)) {
                        if (aupp <= 0 && blow >= 0) {
                                /* (iffy, aupp * blow) */
                                res->low = numunums / 2 + 1;
                                fesetround(FE_UPWARD);
                                res->upp = blur(aupp * blow,
                                                unum, numunums);
                        } else {
                                /* all real numbers */
                                res->low = numunums / 2 + 1;
                                res->upp = numunums / 2 - 1;
                                return;
                        }
                } else if (isinf(aupp) && isinf(blow)) {
                        mul(b, a, res, unum, numunums);
                        return;
                } else if (isinf(alow)) {
                        if (blow >= 0) {
                                /* (iffy, MAX(aupp * blow,
                                 *            aupp * bupp) */
                                res->low = numunums / 2 + 1;
                                fesetround(FE_UPWARD);
                                res->upp = blur(MAX(aupp * blow,
                                                    aupp * bupp),
                                                unum, numunums);
                        } else if (bupp <= 0) {
                                /* (MIN(aupp * blow, aupp * bupp),
                                 *  iffy) */
                                fesetround(FE_DOWNWARD);
                                res->low = blur(MIN(aupp * blow,
                                                    aupp * bupp),
                                                unum, numunums);
                                res->upp = numunums / 2 - 1;
                        } else {
                                /* all real numbers */
                                res->low = numunums / 2 + 1;
                                res->upp = numunums / 2 - 1;
```

```
                                return;
                        }
                } else if (isinf(aupp)) {
                        if (blow >= 0) {
                                /* (MIN(alow * blow, aupp * bupp),
                                 *  iffy) */
                                fesetround(FE_DOWNWARD);
                                res->low = blur(MIN(alow * blow,
                                                    alow * bupp),
                                                unum, numunums);
                                res->upp = numunums / 2 - 1;
                        } else if (bupp <= 0) {
                                /* (iffy, MAX(alow * blow,
                                 *            alow * bupp) */
                                res->low = numunums / 2 + 1;
                                fesetround(FE_UPWARD);
                                res->upp = blur(MAX(alow * blow,
                                                    alow * bupp),
                                                unum, numunums);
                        } else {
                                /* all real numbers */
                                res->low = numunums / 2 + 1;
                                res->upp = numunums / 2 - 1;
                                return;
                        }
                } else if (isinf(blow) || isinf(bupp)) {
                        mul(b, a, res, unum, numunums);
                } else {
                        /* (MIN(C), MAX(C)) */
                        fesetround(FE_DOWNWARD);
                        res->low = blur(MIN(MIN(alow * blow,
                                                alow * bupp),
                                            MIN(aupp * blow,
                                                aupp * bupp)),
                                        unum, numunums);
                        fesetround(FE_UPWARD);
                        res->upp = blur(MAX(MAX(alow * blow,
                                                alow * bupp),
                                            MAX(aupp * blow,
                                                aupp * bupp)),
                                        unum, numunums);
                }
        } else if (!isnan(av) && !isnan(bv)) {
                /*
```

```
                 * a point, b point
                 */
                if ((isinf(av) && (fabs(bv) <= DBL_EPSILON *
                                   fabs(bv) || isinf(bv))) ||
                    (isinf(bv) && (fabs(av) <= DBL_EPSILON *
                                   fabs(av) || isinf(av)))) {
                        /* all extended real numbers */
                        res->low = 0;
                        res->upp = numunums - 1;
                        return;
                } else {
                        fesetround(FE_DOWNWARD);
                        res->low = blur(av * bv, unum, numunums);
                        fesetround(FE_UPWARD);
                        res->upp = blur(av * bv, unum, numunums);
                }
        } else if (!isnan(av) && isnan(bv)) {
                /*
                 * a point, b interval
                 */
                bupp = unum[UCLAMP(b, +1)].val;
                blow = unum[UCLAMP(b, -1)].val;

                if (isinf(av)) {
                        if (isinf(blow)) {
                                if (bupp < 0) {
                                        /* infinity */
                                        res->low = numunums / 2;
                                        res->upp = numunums / 2;
                                        return;
                                } else {
                                        /* all extended real
                                         * numbers */
                                        res->low = 0;
                                        res->upp = numunums - 1;
                                        return;
                                }
                        } else if (isinf(bupp)) {
                                if (blow > 0) {
                                        /* infinity */
                                        res->low = numunums / 2;
                                        res->upp = numunums / 2;
                                        return;
                                } else {
```

```
                                        /* all extended real
                                         * numbers */
                                        res->low = 0;
                                        res->upp = numunums - 1;
                                        return;
                                }
                        } else if ((blow < 0) == (bupp < 0)) {
                                /* infinity */
                                res->low = numunums / 2;
                                res->upp = numunums / 2;
                                return;
                        } else {
                                /* all extended real numbers */
                                res->low = 0;
                                res->upp = numunums - 1;
                                return;
                        }
                } else {
                        /* (MIN(av * blow, av * bupp),
                         *  MAX(av * blow, av * bupp)) */
                        fesetround(FE_DOWNWARD);
                        res->low = blur(MIN(av * blow,
                                            av * bupp),
                                        unum, numunums);
                        fesetround(FE_UPWARD);
                        res->upp = blur(MAX(av * blow,
                                            av * bupp),
                                        unum, numunums);
                }
        } else if (isnan(av) && !isnan(bv)) {
                /*
                 * a interval, b point
                 */
                mul(b, a, res, unum, numunums);
                return;
        }

        if (isnan(av) || isnan(bv)) {
                /* we had an open interval in our calculation
                 * and need to check if res->upp or res->low
                 * are a point. If this is the case, we have
                 * to round it down to respect the openness
                 * of the real interval */
                if (!isnan(unum[res->low].val)) {
```

```
                        res->low = UCLAMP(res->low, +1);
                }
                if (!isnan(unum[res->upp].val)) {
                        res->upp = UCLAMP(res->upp, -1);
                }
        }
}

static void
gentable(char *name, void (*f)(size_t, size_t, struct _unumrange *,
    struct _unum *, size_t), struct _unum *unum, size_t numunums)
{
        struct _unumrange res;
        size_t s, z;

        printf("\nstruct␣_unumrange␣%stable[]␣=␣{\n", name);

        for (z = 0; z < numunums; z++) {
                putc('\t', stdout);

                for (s = 0; s <= z; s++) {
                        f(s, z, &res, unum, numunums);
                        printf("%s{␣%zd,␣%zd␣},", s ? "␣" : "",
                               res.low, res.upp);
                }

                fputs("\n", stdout);
        }

        fputs("};\n", stdout);
}

void
ulog(size_t u, struct _unumrange *res, struct _unum *unum,
    size_t numunums)
{
        double uv, ulow, uupp;

        uv = unum[u].val;

        if (isnan(uv)) {
                ulow = unum[UCLAMP(u, -1)].val;
                uupp = unum[UCLAMP(u, +1)].val;

```

```
                res->low = blur(log(ulow), unum, numunums);
                res->upp = blur(log(uupp), unum, numunums);
        } else {
                res->low = res->upp = blur(log(uv), unum, numunums);
        }
}

static void
genfunctable(char *name, void (*f)(size_t, struct _unumrange *,
    struct _unum *, size_t), struct _unum *unum, size_t numunums)
{
        struct _unumrange res;
        size_t u;

        printf("\nstruct␣_unumrange␣%stable[]␣=␣{\n", name);

        for (u = 0; u <= numunums / 2; u++) {
                f(u, &res, unum, numunums);
                printf("\t{␣%zd,␣%zd␣},\n", res.low, res.upp);
        }

        fputs("};\n", stdout);
}

static void
genunums(struct _latticep *lattice, size_t latticesize,
         struct _unum *unum, size_t numunums)
{
        size_t off;
        ssize_t i;

        off = 0;

        /* 0 */
        unum[off].val = 0.0;
        unum[off].name = "0";
        off++;
        unum[off].val = NAN;
        unum[off].name = NULL;
        off++;

        /* (0,1) */
        for (i = latticesize - 1; i >= 0; i--, off++) {
                unum[off].val = 1 / lattice[i].val;
```

```
                if (lattice[i].name[0] == '/') {
                        unum[off].name = lattice[i].name + 1;
                } else {
                        /* add '/' prefix */
                        if (!(unum[off].name =
                              malloc(strlen(lattice[i].name) + 2))) {
                                fprintf(stderr, "out␣of␣memory\n");
                                exit(1);
                        }
                        strcpy(unum[off].name + 1, lattice[i].name);
                        unum[off].name[0] = '/';
                }
                off++;
                unum[off].val = NAN;
                unum[off].name = NULL;
        }

        /* 1 */
        unum[off].val = 1.0;
        unum[off].name = "1";
        off++;
        unum[off].val = NAN;
        unum[off].name = NULL;
        off++;

        /* (1,INF) */
        for (i = 0; i < latticesize; i++, off++) {
                unum[off].val = lattice[i].val;
                unum[off].name = lattice[i].name;
                off++;
                unum[off].val = NAN;
                unum[off].name = NULL;
        }

        /* INF */
        unum[off].val = INFINITY;
        unum[off].name = "\u221E";
        off++;
        unum[off].val = NAN;
        unum[off].name = NULL;
        off++;

        /* (INF,-1) */
        for (i = latticesize - 1; i >= 0; i--, off++) {
```

```
        unum[off].val = -lattice[i].val;
        if (!(unum[off].name =
           malloc(strlen(lattice[i].name) + 2))) {
            fprintf(stderr, "out␣of␣memory\n");
            exit(1);
        }
        strcpy(unum[off].name + 1, lattice[i].name);
        unum[off].name[0] = ’-’;
        off++;
        unum[off].val = NAN;
        unum[off].name = NULL;
    }

    /* -1 */
    unum[off].val = -1.0;
    unum[off].name = "-1";
    off++;
    unum[off].val = NAN;
    unum[off].name = NULL;
    off++;

    /* (-1, 0) */
    for (i = 0; i < latticesize; i++, off++) {
        unum[off].val = - 1 / lattice[i].val;
        if (lattice[i].name[0] == ’/’) {
            if (!(unum[off].name =
               strdup(lattice[i].name))) {
                fprintf(stderr, "out␣of␣memory\n");
                exit(1);
            }
            unum[off].name[0] = ’-’;
        } else {
            /* add ’-/’ prefix */
            if (!(unum[off].name =
               malloc(strlen(lattice[i].name) + 3))) {
                fprintf(stderr, "out␣of␣memory\n");
                exit(1);
            }
            strcpy(unum[off].name + 2, lattice[i].name);
            unum[off].name[0] = ’-’;
            unum[off].name[1] = ’/’;
        }
        off++;
        unum[off].val = NAN;
```

```
                unum[off].name = NULL;
        }
}

void
gendeclattice(struct _latticep **lattice, size_t *latticesize,
              double maximum, int sigdigs)
{
        size_t i, maxlen;
        double c1, c2, curmax;
        char *fmt = "%.*f";

        /*
         * Check prerequisites
         */
        if (sigdigs == 0) {
                fprintf(stderr, "invalid␣number␣of␣"
                        "significant␣digits\n");
        }
        if ((*latticesize == 0) == isinf(maximum)) {
                fprintf(stderr, "gendeclattice:␣accepting␣"
                "only␣one␣parameter␣besides␣number␣of␣"
                "significant␣digits\n");
                exit(1);
        }

        c1 = pow(10, sigdigs) - pow(10, sigdigs - 1);
        c2 = pow(10, -(sigdigs - 1));

        if (*latticesize == 0) {
                /* calculate lattice size until maximum is
                 * contained */
                for (curmax = 0; curmax < maximum; (*latticesize)++) {
                        curmax = (1 + c2 *
                                  (*latticesize % (size_t)c1)) *
                                 pow(10, floor(*latticesize / c1));
                }
        } else { /* isinf(maximum) */
                /* calculate maximum */
                maximum = (1 + c2 *
                           (*latticesize % (size_t)c1)) *
                          pow(10, floor(*latticesize / c1));
        }

```

```
        /*
         * Generate lattice
         */
        if (!(*lattice = malloc(sizeof(struct _latticep) *
                                *latticesize))) {
                fprintf(stderr, "out␣of␣memory\n");
                exit(1);
        }
        maxlen = snprintf(NULL, 0, fmt, sigdigs - 1, maximum) + 1;
        for (i = 0; i < *latticesize; i++) {
                (*lattice)[i].val = (1 + c2 *
                                      ((i + 1) % (size_t)c1)) *
                                     pow(10, floor((i + 1) / c1));
                if (!((*lattice)[i].name = malloc(maxlen))) {
                        fprintf(stderr, "out␣of␣memory\n");
                        exit(1);
                }
                snprintf((*lattice)[i].name, maxlen, fmt,
                         sigdigs - 1, (*lattice)[i].val);
        }
}

int
main(void)
{
        struct _unum *unum;
        struct _latticep *lattice;
        size_t latticebits, latticesize, numunums;
        ssize_t i;
        int bits;

        /* Generate lattice */
        latticesize = (1 << (UBITS - 3)) - 1;
        gendeclattice(&lattice, &latticesize, INFINITY, DIGITS);

        /*
         * Print unum.h includes
         */
        fprintf(stderr, "#include␣<math.h>\n#include␣<stddef.h>\n"
                "#include␣<stdint.h>\n\n");

        /*
         * Determine number of effective bits used
         */
```

```
        struct {
                int bits;
                char *type;
        } types[] = {
                { 8,  "uint8_t"  },
                { 16, "uint16_t" },
                { 32, "uint32_t" },
                { 64, "uint64_t" },
        };

        numunums = 8 * (latticesize + 1);
        for (bits = 1; bits <= types[LEN(types) - 1].bits; bits++) {
                if (numunums == (1 << bits))
                        break;
        }
        if (bits > types[LEN(types) - 1].bits) {
                fprintf(stderr, "invalid␣number␣of␣lattice"
                        "points\n");
                return 1;
        }

        /*
         * Determine type needed to store the unum
         */
        for (i = 0; i < LEN(types); i++) {
                if (types[i].bits >= bits)
                        break;
        }
        if (i == LEN(types)) {
                fprintf(stderr, "cannot␣fit␣bits␣into␣system"
                        "types\n");
                return 1;
        }

        /*
         * Print list of preliminary unum.h definitions
         */
        fprintf(stderr, "typedef␣%s␣unum;\n#define␣ULEN␣%d\n"
                "#define␣NUMUNUMS␣%zd\n", types[i].type, bits,
                numunums);

        fprintf(stderr, "#define␣UCLAMP(i,␣off)␣(((((off␣<␣0)␣&&␣(i)␣<"
                "-off)␣?␣\\\n\tNUMUNUMS␣-␣((-off␣-␣(i))␣%%␣NUMUNUMS)␣:"
                "\\\n\t((off␣>␣0)␣&&␣(i)␣+␣off␣>␣NUMUNUMS␣-␣1)␣?␣\\\n\t"
```

```
                "((i)␣+␣off␣%%␣NUMUNUMS)␣%%␣NUMUNUMS␣:␣(i)␣+␣off))␣%%␣"
                "NUMUNUMS)\n\n");

        fprintf(stderr, "typedef␣struct␣{\n\tuint8_t␣data[%d];\n}␣SORN;\n",
                (1 << bits) / 8);

        fprintf(stderr, "\nvoid␣uadd(SORN␣*,␣SORN␣*);\n"
                "void␣usub(SORN␣*,␣SORN␣*);\n"
                "void␣umul(SORN␣*,␣SORN␣*);\n"
                "void␣udiv(SORN␣*,␣SORN␣*);\n"
                "void␣uneg(SORN␣*);\n"
                "void␣uinv(SORN␣*);\n"
                "void␣uabs(SORN␣*);\n\n"
                "void␣ulog(SORN␣*);\n\n"
                "void␣uemp(SORN␣*);\n"
                "void␣uset(SORN␣*,␣SORN␣*);\n"
                "void␣ucut(SORN␣*,␣SORN␣*);\n"
                "void␣uuni(SORN␣*,␣SORN␣*);\n"
                "int␣uequ(SORN␣*,␣SORN␣*);\n"
                "int␣usup(SORN␣*,␣SORN␣*);\n\n"
                "void␣uint(SORN␣*,␣double,␣double);\n"
                "void␣uout(SORN␣*);\n");

        /*
         * Generate unums
         */
        if (!(unum = malloc(sizeof(struct _unum) * numunums))) {
                fprintf(stderr, "out␣of␣memory\n");
                return 1;
        }
        genunums(lattice, latticesize, unum, numunums);

        /*
         * Print table.c includes
         */
        printf("#include␣\"table.h\"\n");

        /*
         * Print list of unums
         */
        printunums(unum, numunums);

        /*
         * Generate and print tables
```

```
         */
        gentable("add", add, unum, numunums);
        gentable("mul", mul, unum, numunums);

        /*
         * Generate function tables
         */
        genfunctable("log", ulog, unum, numunums);

        return 0;
}
```

B.2.2. table.h

```
#include "unum.h"

struct _unumrange {
        unum a;
        unum b;
};

struct _unum {
        double val;
        char *name;
};

extern struct _unum unums[];
extern struct _unumrange addtable[];
extern struct _unumrange multable[];
extern struct _unumrange logtable[];
```

B.2.3. unum.c

```
#include <float.h>
#include <math.h>
#include <stdio.h>

#include "table.h"

#undef MAX
#define MAX(x,y)  ((x) > (y) ? (x) : (y))

```

```
static size_t
blur(double val)
{
        size_t i;

        /* infinity is infinity */
        if (isinf(val)) {
                return NUMUNUMS / 2;
        }

        for (i = 0; i < NUMUNUMS; i++) {
                /* equality within relative epsilon */
                if (isfinite(unums[i].val) && fabs(unums[i].val - val) <=
                    DBL_EPSILON * MAX(fabs(unums[i].val), fabs(val))) {
                        return i;
                }

                /* in range */
                if (isnan(unums[i].val) &&
                    val < unums[UCLAMP(i, +1)].val &&
                    val > unums[UCLAMP(i, -1)].val) {
                        return i;
                }
        }

        /* negative or positive range outshot */
        return (val < 0) ? (NUMUNUMS / 2 + 1) : (val > 0) ?
               (NUMUNUMS / 2 - 1) : 0;
}

static void
_sornaddrange(SORN *s, unum lower, unum upper)
{
        unum u;
        size_t i, j;
        int first;

        for (first = 1, u = lower; u != UCLAMP(upper, +1) ||
             (first && lower == UCLAMP(upper, +1));
            u = UCLAMP(u, +1)) {
                first = 0;
                i = u / (sizeof(*s->data) * 8);
                j = u % (sizeof(*s->data) * 8);

```

```
                s->data[i] |= (1 << (sizeof(*s->data) * 8 - 1 - j));
        }
}

static unum
_unumnegate(unum u)
{
        return UCLAMP(NUMUNUMS, -u);
}

static unum
_unuminvert(unum u)
{
        return _unumnegate(UCLAMP(u, +(NUMUNUMS / 2)));
}

static unum
_unumabs(unum u)
{
        return (u > NUMUNUMS / 2) ? _unumnegate(u) : u;
}

static void
_sornop(SORN *a, SORN *b, struct _unumrange table[],
        unum (*mod)(unum))
{
        unum u, v, low, upp;
        size_t i, j, m, n;
        static SORN res;

        /* empty result SORN */
        for (i = 0; i < sizeof(res.data); i++) {
                res.data[i] = 0;
        }

        for (i = 0; i < sizeof(a->data); i++) {
                for (j = 0; j < sizeof(*a->data) * 8; j++) {
                        if (!(a->data[i] & (1 << (sizeof(*a->data) * 8 -
                                                    1 - j)))) {
                                continue;
                        }
                        /* unum u = (so * i + j) is in the first set */
                        u = sizeof(*a->data) * 8 * i + j;
                        for (m = 0; m < sizeof(b->data); m++) {
```

```
                                for (n = 0; n < sizeof(*b->data) * 8;
                                     n++) {
                                        if (!(b->data[m] & (1 <<
                                            (sizeof(*b->data) * 8 - 1 -
                                             n)))) {
                                                continue;
                                        }
                                        /* unum v = (so * m + n) is in
                                         * the second set */
                                        v = sizeof(*b->data) * 8 * m + n;

                                        /*
                                         * compare struct pointers to
                                         * identify dependent arguments
                                         * and in this case only do
                                         * pairwise operations
                                         */
                                        if (a == b && u != v)
                                                continue;

                                        /* apply an optional modifier
                                         * after the dependency check */
                                        if (mod) {
                                                v = mod(v);
                                        }

                                        /* get bounds from table;
                                         * according to gauß 1 + 2 + 3
                                         * ... + n = (n * (n + 1)) / 2,
                                         * used to traverse triangle
                                         * array */
                                        if (u <= v) {
                                                low = table[((size_t)v *
                                                              (v + 1)) /
                                                             2 + u].a;
                                                upp = table[((size_t)v *
                                                              (v + 1)) /
                                                             2 + u].b;
                                        } else {
                                                low = table[((size_t)u *
                                                              (u + 1)) /
                                                             2 + v].a;
                                                upp = table[((size_t)u *
                                                              (u + 1)) /
```

```
                                                                2 + v].b;

                                        }
                                        _sornaddrange(&res, low, upp);
                                }
                        }
                }
        }

        /* write result to first operand */
        for (i = 0; i < sizeof(a->data); i++) {
                a->data[i] = res.data[i];
        }
}

static void
_sornmod(SORN *s, unum (mod)(unum))
{
        SORN res;
        unum u;
        size_t i, j, k, l;

        for (i = 0; i < sizeof(res.data); i++) {
                res.data[i] = 0;
        }

        for (i = 0; i < sizeof(s->data); i++) {
                for (j = 0; j < sizeof(*s->data) * 8; j++) {
                        if (!(s->data[i] & (1 <<
                              (sizeof(*s->data) * 8 - 1 - j)))) {
                                continue;
                        }
                        /* unum u = (so * i + j) is in the set */
                        u = sizeof(*s->data) * 8 * i + j;
                        u = mod(u);

                        k = u / (sizeof(*s->data) * 8);
                        l = u % (sizeof(*s->data) * 8);
                        res.data[k] |= (1 << (sizeof(*res.data) *
                                              8 - 1 - l));
                }
        }

        /* write result to operand */
```

```
        for (i = 0; i < sizeof(s->data); i++) {
                s->data[i] = res.data[i];
        }
}

void
uadd(SORN *a, SORN *b)
{
        _sornop(a, b, addtable, NULL);
}

void
usub(SORN *a, SORN *b)
{
        _sornop(a, b, addtable, _unumnegate);
}

void
umul(SORN *a, SORN *b)
{
        _sornop(a, b, multable, NULL);
}

void
udiv(SORN *a, SORN *b)
{
        _sornop(a, b, multable, _unuminvert);
}

void
uneg(SORN *s)
{
        _sornmod(s, _unumnegate);
}

void
uinv(SORN *s)
{
        _sornmod(s, _unuminvert);
}

void
uabs(SORN *s)
{
```

```
        _sornmod(s, _unumabs);
}

void
ulog(SORN *s)
{
        unum u;
        size_t i, j;
        static SORN res;

        /* empty result SORN */
        for (i = 0; i < sizeof(res.data); i++) {
                res.data[i] = 0;
        }

        for (i = 0; i < sizeof(s->data); i++) {
                for (j = 0; j < sizeof(*s->data) * 8; j++) {
                        if (!(s->data[i] & (1 << (sizeof(*s->data) * 8 -
                                                  1 - j)))) {
                                continue;
                        }
                        /* unum u = (so * i + j) is in the set */
                        u = sizeof(*s->data) * 8 * i + j;

                        /* is SORN negative? not defined */
                        if (u > NUMUNUMS / 2) {
                                _sornaddrange(&res, 0, NUMUNUMS - 1);
                                goto done;
                        }

                        /* read the table and apply ranges */
                        _sornaddrange(&res, logtable[u].a,
                                      logtable[u].b);
                }
        }
done:
        /* write result to operand */
        for (i = 0; i < sizeof(s->data); i++) {
                s->data[i] = res.data[i];
        }
}

void
uemp(SORN *s)
```

```
{
	size_t i;

	for (i = 0; i < sizeof(s->data); i++) {
		s->data[i] = 0;
	}
}

void
uset(SORN *a, SORN *b)
{
	size_t i;

	for (i = 0; i < sizeof(a->data); i++) {
		a->data[i] = b->data[i];
	}
}

void
ucut(SORN *a, SORN *b)
{
	size_t i;

	for (i = 0; i < sizeof(a->data); i++) {
		a->data[i] = a->data[i] & b->data[i];
	}
}

void
uuni(SORN *a, SORN *b)
{
	size_t i;

	for (i = 0; i < sizeof(a->data); i++) {
		a->data[i] = a->data[i] | b->data[i];
	}
}

int
uequ(SORN *a, SORN *b)
{
	size_t i;

	for (i = 0; i < sizeof(a->data); i++) {
```

```
                if (a->data[i] != b->data[i]) {
                        return 0;
                }
        }

        return 1;
}

int
usup(SORN *a, SORN *b)
{
        ucut(a, b);

        return uequ(a, b);
}

void
uint(SORN *s, double lower, double upper)
{
        _sornaddrange(s, blur(lower), blur(upper));
}

void
uout(SORN *s)
{
        unum loopstart, u;
        size_t i, j;
        int active, insorn, loop2run;

        loop2run = 0;
        for (active = 0, i = sizeof(s->data) / 2; i < sizeof(s->data);
             i++) {
loop1start:
                for (j = 0; j < sizeof(*s->data) * 8; j++) {
                        u = sizeof(*s->data) * 8 * i + j;
                        insorn = s->data[i] & (1 << (sizeof(*s->data) *
                                                       8 - 1 - j));
                        if (!active && insorn) {
                                /* print the opening of a closed
                                 * subset */
                                active = 1;
                                if (unums[u].name) {
                                        printf("[%s,", unums[u].name);
                                } else {
```

```
                                        printf("(%s,", unums[UCLAMP(u,
                                                       -1)].name);
                                }
                        } else if (active && !insorn) {
                                /* print the closing of a closed
                                 * subset */
                                active = 0;
                                if (unums[UCLAMP(u, -1)].name) {
                                        printf("%s]␣", unums[UCLAMP(u,
                                                       -1)].name);
                                } else {
                                        printf("%s)␣", unums[u].name);
                                }
                        }
                }
                if (loop2run) {
                        goto loop2end;
                }
        }

        loop2run = 1;
        for (i = 0; i < sizeof(s->data) / 2; i++) {
                goto loop1start;
loop2end:
                ;
        }

        if (active) {
                printf("\u221E)");
        }
}
```

B.2.4. config.mk

```
UBITS = 12
DIGITS = 2
```

B.2.5. Makefile

```
include config.mk

all: libunum.a
```

```

libunum.a: table.o unum.o
	ar rcs libunum.a table.o unum.o

unum.o: unum.c
	cc -c unum.c -lm

table.o: table.c
	cc -c table.c

table.c: gen
	./gen 2> unum.h 1> table.c

gen: gen.c config.mk
	cc -o gen -DUBITS=${UBITS} -DDIGITS=${DIGITS} gen.c -lm

%: %.c libunum.a
	cc $^ -o $@

clean:
	rm -f gen table.c unum.h table.o unum.o libunum.a
```

B.3. Unum Problems

These programs expect libunum.a and unum.h in the current directory at compile time. It is recommended to create symbolic links to the toolbox directory given both are generated dynamically there and thus subject to change.

The environment parametres for the decade lattice are set in `config.mk` (see Listing B.2.4).

B.3.1. euler.c

```
#include <stdio.h>

#include "unum.h"

void
factorial(SORN *s, int f)
{
	SORN tmp;
	int i;

	uemp(s);
```

```
        uint(s, 1, 1);

        for (i = f; i > 1; i--) {
                uemp(&tmp);
                uint(&tmp, i, i);
                umul(s, &tmp);
        }
}

int
main(void)
{
        SORN e, tmp;
        int i;

        uemp(&e);
        uint(&e, 1, 1);
        uout(&e);
        putchar('\n');

        for (i = 1; i <= 20; i++) {
                factorial(&tmp, i);
                uinv(&tmp);
                uadd(&e, &tmp);
                uout(&e);
                putchar('\n');
        }

        return 0;
}
```

B.3.2. devil.c

```
#include <stdio.h>

#include "unum.h"

int
main(void)
{
        SORN a, b, c, tmp1, tmp2, tmp3;
        int n;

```

```
        uemp(&a);
        uemp(&b);
        uemp(&c);

        uint(&a, 2, 2);
        uint(&b, -4, -4);

        for (n = 2; n <= 25; n++) {
                if (n > 2) {
                        uset(&a, &b);
                        uset(&b, &c);
                }

                uemp(&tmp1);
                uint(&tmp1, 111, 111);

                uemp(&tmp2);
                uint(&tmp2, 1130, 1130);
                udiv(&tmp2, &b);
                usub(&tmp1, &tmp2);

                uemp(&tmp2);
                uint(&tmp2, 3000, 3000);
                uset(&tmp3, &b);
                umul(&tmp3, &a);

                udiv(&tmp2, &tmp3);
                uadd(&tmp1, &tmp2);

                uset(&c, &tmp1);

                printf("U_%d␣=␣", n);
                uout(&c);
                putchar('\n');
        }

        return 0;
}
```

B.3.3. bank.c

```
#include <math.h>
#include <stdio.h>
```

```

#include "unum.h"

int
main(void)
{
	SORN a, tmp;
	int y;

	uemp(&a);
	uint(&a, M_E - 1, M_E - 1);

	for (y = 1; y <= 25; y++) {
		uemp(&tmp);
		uint(&tmp, y, y);
		umul(&a, &tmp);

		uemp(&tmp);
		uint(&tmp, 1, 1);
		usub(&a, &tmp);

		printf("year␣%2d:␣", y);
		uout(&a);
		putchar('\n');
	}

	return 0;
}
```

B.3.4. spike.c

```
#include <stdio.h>

#include "unum.h"

#define NUMPOINTS (10)
#define POLE (4.0/3.0)

int
main(void)
{
	SORN res, tmp;
	size_t i, j;
```

```
        unum pole, u;

        /* get the unum containing the POLE */
        uemp(&res);
        uint(&res, POLE, POLE);
        for (i = 0; i < sizeof(res.data); i++) {
                if (res.data[i]) {
                        for (j = 0; j < sizeof(res.data[i]) * 8; j++) {
                                if ((1 << (8 - j - 1)) & res.data[i]) {
                                        break;
                                }
                        }
                        pole = 8 * i + j;
                        break;
                }
        }

        for (u = UCLAMP(pole, -NUMPOINTS); u <= UCLAMP(pole,
             +NUMPOINTS); u++) {
                /* fill res just with the single u */
                uemp(&res);
                res.data[u / 8] = 1 << (8 - (u % 8) - 1);
                uout(&res);
                printf("␣|->␣");

                /* calculate F(res) */
                uneg(&res);

                uemp(&tmp);
                uint(&tmp, 1, 1);
                uadd(&res, &tmp);

                uemp(&tmp);
                uint(&tmp, 3, 3);
                umul(&res, &tmp);

                uemp(&tmp);
                uint(&tmp, 1, 1);
                uadd(&res, &tmp);

                uabs(&res);
                ulog(&res);
                uout(&res);
                putchar('\n');
```

```
        }

        return 0;
}
```

B.3.5. Makefile

```
PROBLEMS = euler devil bank spike

all: $(PROBLEMS)

%: %.c
        cc -o $@ $^ libunum.a

clean:
        rm -rf $(PROBLEMS)
```

B.4. License

This ISC license applies to all code listings in Chapter B.

```
Copyright © 2016, Laslo Hunhold

Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted, provided that the above
copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

Bibliography

[Gus15] John L. Gustafson. *The End of Error: Unum Computing*. Computational Science Series. Chapman & Hall/CRC, Boca Raton, Florida, USA, 1 edition, February 2015. ISBN 9781482239867.

[Gus16a] John L. Gustafson. An energy-efficient and massively parallel approach to valid numerics. ICRAR Seminar, `http://www.johngustafson.net/presentations/UnumArithmetic-ICRARseminar.pdf`, June 2, 2016. Accessed: 2016-08-31.

[Gus16b] John L. Gustafson. A radical approach to computation with real numbers. `http://www.johngustafson.net/pubs/RadicalApproach.pdf`, May 2016. Accessed: 2016-08-31.

[IEE85] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. New York City, NY, USA, August 12, 1985. 20 pp.

[IEE08] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. New York City, NY, USA, August 29, 2008. 58 pp.

[IEE15] IEEE Task P1788. *IEEE 1788-2015, Standard for Interval Arithmetic*. New York City, NY, USA, June 11, 2015. 97 pp.

[Kah06] William Morton Kahan. How futile are mindless assessments of roundoff in floating-point computation? `https://people.eecs.berkeley.edu/~wkahan/Mindless.pdf`, January 2006. Accessed: 2016-08-31.

[Kow14] Hans-Joachim Kowalsky. *Topological Spaces*. Academic Press, New York City, NY, USA, May 12, 2014. ISBN 9781483265247. 296 pp.

[MBdD$^+$10] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, Boston, MA, USA, 1 edition, December 2010. ISBN 9780817647049.

[MKC09] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1 edition, 2009. ISBN 9780898716696. 223 pp.

[MKŠ+06] Rafi L. Muhanna, Vladik Kreinovich, Pavel Šolín, Jack Chessa, Roberto Araiza, and Gang Xiang. Interval finite element methods: New directions. In Rafi L. Muhanna and Robert L. Mullen, editors, *Modeling Errors and Uncertainty in Engineering Computations*, pages 229–243. NSF Workshop on Reliable Engineering Computing, REC 2006, Atlanta, GA, USA, January 1, 2006. ISBN 044486377X.

[Moo67] Ramon E. Moore. *Interval Analysis.* Prentice-Hall Series in Automatic Computation. Prentice Hall, Upper Saddle River, NJ, USA, 1 edition, January 1967. ISBN 9780134768533. 145 pp.

[Moo79] Ramon E. Moore. *Methods and Applications of Interval Analysis.* Studies in Applied and Numerical Mathematics. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1 edition, 1979. ISBN 9780898711615. 190 pp.

[PdCMBL96] A. Pinto da Costa, J. A. C. Martins, F. Branco, and J. L. Lilien. Oscillations of bridge stay cables induced by periodic motions of deck and/or towers. *Journal of Engineering Mechanics*, 122(7):613–622, July 1996.

[Ran82] Brian Randell. From Analytical Engine to electronic digital computer: The contributions of Ludgate, Torres, and Bush. *Annals of the History of Computing*, 4(4):327–341, October/December 1982.

[Rei82] C. Reinsch. A synopsis of interval arithmetic for the designer of programming languages. In John Ker Reid, editor, *The Relationship Between Numerical Computation and Programming Languages*, pages 85–97. IFIP Working Group 2.5–Mathematical Software, IFIP Technical Committee 2–Programming, Elsevier Science Ltd, New York City, NY, USA, April 1982. ISBN 044486377X.

[Roj98] Raúl Rojas. *Die Rechenmaschinen von Konrad Zuse.* Springer-Verlag, Berlin Heidelberg, 1 edition, January 1, 1998. ISBN 9783642719455. 236 pp.

[Ser15] Yaroslav D. Sergeyev. Computations with grossone-based infinities. In Cristian S. Calude and Michael J. Dinneen, editors, *Unconventional Computation and Natural Computation*, volume 9252 of *Lecture Notes in Computer Science*, pages 89–106. UCNC 2015, Springer International Publishing, Cham, Switzerland, 2015. ISBN 9783319218182.

[TF95] George B. Thomas and Ross L. Finney. *Calculus and Analytic Geometry.* Addison Wesley, Boston, MA, USA, 9 edition, August 1995. ISBN 9780201531749.

Eigenständigkeitserklärung

Hiermit bestätige ich, daß ich die vorliegende Arbeit selbstständig verfaßt und keine anderen als die angegebenen Hilfsmittel verwendet habe.
Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Laslo Hunhold