Universal: Reliable, Reproducible, and Energy-Efficient Numerics

E. Theodore L. Omtzigt[1](✉) and James Quinlan[2]

[1] Stillwater Supercomputing, Inc., El Dorado Hills, CA 95762, USA
tomtzigt@stillwater-sc.com
[2] University of New England, Biddeford, ME 04005, USA
jquinlan@une.edu
https://stillwater-sc.com

Abstract. *Universal* provides a collection of arithmetic types, tools, and techniques for performant, reliable, reproducible, and energy-efficient algorithm design and optimization. The library contains a full spectrum of custom arithmetic data types ranging from memory-efficient fixed-size arbitrary precision integers, fixed-points, regular and tapered floating-points, logarithmic, faithful, and interval arithmetic, to adaptive precision integer, decimal, rational, and floating-point arithmetic. All arithmetic types share a common control interface to set and query bits to simplify numerical verification algorithms. The library can be used to create mixed-precision algorithms that minimize the energy consumption of essential algorithms in embedded intelligence and high-performance computing. *Universal* contains command-line tools to help visualize and interrogate the encoding and decoding of numeric values in all the available types. Finally, *Universal* provides error-free transforms for floating-point and reproducible computation and linear algebra through user-defined rounding techniques.

Keywords: Mixed-precision algorithm · Energy-efficient arithmetic · Reliable computing · Reproducibility

1 Introduction

The proliferation of computing across more diverse use cases drives application and algorithm innovation in more varied goals. Performance continues to be a key driver in product differentiation, but energy efficiency is essential for embedded and edge computing. Moreover, reliable and reproducible computing is paramount in safety applications such as autonomous vehicles.

In embedded systems, energy is at a premium, and the application must deliver a solution within a strict power constraint to be viable. In hyperscaled

Developed by open-source developers, and supported and maintained by Stillwater Supercomputing Inc.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
J. Gustafson and V. Dimitrov (Eds.): CoNGA 2022, LNCS 13253, pp. 100–116, 2022.
https://doi.org/10.1007/978-3-031-09779-9_7

cloud data centers, the cost of electricity has overtaken the acquisition cost of IT equipment, making energy efficiency even an economic driver for the cloud.

The reproducibility of computational results is essential to applications that impact human safety and collaboration. Reproducible computation is required for forensic analysis to explain a recorded failure of an autonomous vehicle. Collaborative projects that leverage computational science are more efficient when two different research groups can reproduce simulation results on different platforms. *Universal* provides the deferred rounding machinery to implement reproducibility. Lastly, numerically sensitive results require verification or quality assertions. Reliable computing provides such guarantees on accuracy or bounding boxes of error.

The *Universal* library offers custom arithmetic types and utilities for optimizing energy efficiency, performance, reproducibility, and reliability of computational systems. Once the arithmetic solution has been found, *Universal* provides a seamless transition to create and leverage custom compute engines to accelerate the execution of the custom arithmetic. And finally, *Universal* provides a unified mechanism to extend other language environments, such as MATLAB/Simulink, Python, or Julia, with validated and verified custom arithmetic types.

2 Background and Motivation

Deep learning algorithms transform applications that classify and characterize patterns in vision, speech, language, and optimal control. This so-called Software 2.0 transformation uses data and computation to synthesize and manage the behavior and capability of the application. In deep learning applications, computational demand is high, and data supplied is varied, making performance and energy efficiency paramount for success. The industry has responded with a proliferation of custom hardware accelerators running energy-conserving numeric systems.

These hardware accelerators need to be integrated into embedded, network, and cloud ecosystems. For example, Google TPUs [16], and Intel CPUs [15] support a type called a brain float, which is a 16-bit floating-point format that truncates the lower 16-bits of a standard IEEE-754 single-precision float. NVIDIA, on the other hand, implements their unique data type, the TensorFloat-32 (TF32) [17], which is a 19-bit format with 8 bits encoding the exponent, and 10 bits encoding the mantissa.

This proliferation of vendor-specific types creates demand for solutions that enable software designers to create, run, test, and deploy applications that take advantage of custom arithmetic while, at the same time, integrating seamlessly with a broad range of hardware accelerators. Software that can evaluate, adapt, or replace different arithmetic types requires an upfront investment in architecture design and implementation. *Universal* offers an environment where anyone can deploy custom arithmetic types while maintaining complete flexibility to adapt to better hardware from a different vendor when available.

As the research and development community learns more about the computational dynamics of Software 2.0 applications, novel number system representations will be invented to enable new application capabilities. For example, massive multiple-input multiple-output (MIMO) systems in cellular networks will benefit from optimized arithmetic [21]. Convolutional Neural Networks are showing attributes that favor logarithmic number systems [27]. Safety systems require arithmetic that is reproducible and numeric algorithms that are reliable [22]. Large scale high-performance computing applications that model physical phenomena leverage continuity constraints to compress fields of metrics to lower power consumption and maximize memory performance [13,18] This list of custom number systems and their arithmetic type representations will only grow, igniting a renewed focus on efficiency of representation and computation.

Numerical environments such as `Boost MultiPrecision` [19], `MPFR` [5], and `GMP` [7] have been focused on providing extensions to IEEE-754. They are not tailored to providing new arithmetic types and encodings as required for improving Software 2.0 applications. In contrast, *Universal* is purposefully designed to offer and integrate new arithmetic types for the emerging applications in embedded intelligence, mobile, and cloud computing.

Universal started in 2017 as a hardware verification library for the emerging posit standard [9]. It provided a hardware model of a bit-level implementation of arbitrary configuration posits, parameterized as `posit<nbits,es>` and presented as a plug-in arithmetic type for C++ linear-algebra libraries [6,25,26]. Since then, *Universal* has grown into a research and development platform for multi-precision algorithm optimization and numerical analysis.

More recently, it has been instrumental in developing applications that exhibit strong cooperation between general-purpose processing on the CPU and special-purpose processing on accelerators. As *Universal* arithmetic types operate with the same encoding and memory layout as the hardware accelerator, applications can use the general-purpose CPU to serialize, prepare, and manage the data structures on behalf of the custom hardware accelerator without the need for conversions to and from native types.

In this paper, we provide a status update to the third edition of the *Universal* library [23]. In Sect. 3, we discuss the various arithmetic types available in *Universal* and where they fit in the set hierarchy representing abstract algebraic number systems. Section 4 discusses the design flow for creating new arithmetic types that have proven to be productive. *Universal* is still very much a hardware/software co-design library, so Sect. 5 describes the standard application programming interface of the arithmetic types in *Universal* that simplify integration into verification and regression test suites. Finally, Sect. 6 demonstrates different algorithm and application framework integration examples. We conclude in Sect. 7 with a summary and future work.

3 Universal Arithmetic Type Organization

Figure 1 shows the cover of the different arithmetic types available in *Universal* relative to the known algebraic sets.

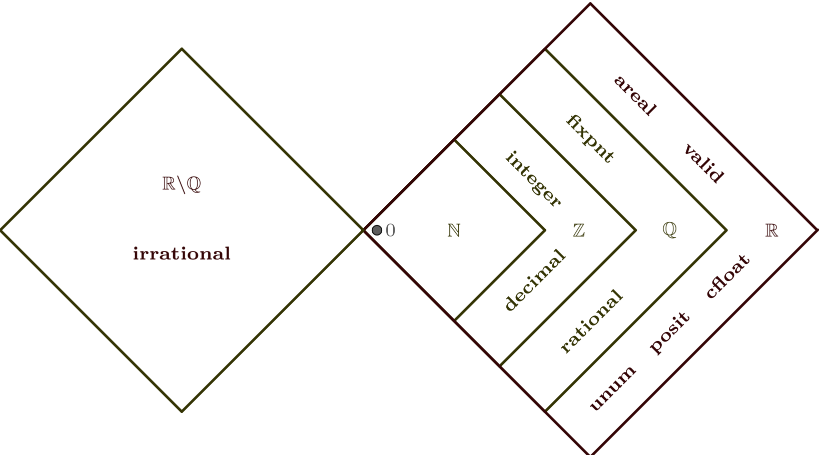

Fig. 1. Abstract algebraic sets and *Universal* arithmetic types

Universal classifies arithmetic types into *fixed* and *adaptive* types. *Fixed* types are arithmetic types that have a fixed memory layout when declared. In contrast, the memory layout of *adaptive* types varies during the computation. Fixed types are intended for energy-efficient, performant, and linear-algebra-focused applications. Adaptive types are more suitable for accuracy and reliable computing investigations.

For the *fixed* arithmetic types, *Universal* strives to offer sizes that are configurable by individual bits as the target are custom hardware implementations in specialized hardware accelerators. The parameterization space of *fixed* arithmetic types is:

1. sampling profile and encoding
2. size in bits
3. dynamic range
4. arithmetic behavior (modulo, saturate, clip etc.)

The most informative example is the classic floating-point type: **cfloat**. It is parameterized in all dimensions:

```
cfloat<nbits, es, BlockType,
       hasSubnormals, hasSupernormals, isSaturating>
```

We will discuss the details in Sect. 3.3.

3.1 Definitions

In *Universal* a custom arithmetic type is defined by a memory layout of the data type, an encoding, and a set of operators that approximate an abstract algebra. For example, a simple algebra on a ring where the data encoding is constrained

to an 8-bit signed integer. Alternatively, a decimal floating-point type that emulates a field with arithmetic operators, addition, subtraction, multiplication, and division $(+, -, \times, \div)$.

Most arithmetic types in *Universal* represent such fields, albeit with a limited range, and arithmetic rules that express how results are handled when they fall outside of the representable range. *Universal* provides a rich set of types that are very small and are frequently found in hardware designs to maximize silicon efficiency.

3.2 Energy Efficiency, Performance, Accuracy, Reliability, and Reproducibility

The energy consumption of a digital logic circuit is proportional to the silicon area occupied. For an arithmetic type over a field, the silicon area of its Arithmetic Logic Unit (ALU) circuit is directly proportional to the square of the size of the encoding. Therefore, the constraint on the range covered by an arithmetic type is a crucial design parameter for energy efficiency optimization.

However, in sub-micron chip manufacturing technology, the energy consumption of data movement is more significant [14]. In 45 nm technology, an 8-bit addition consumes about 0.03pJ, and a 32-bit floating-point multiply consumes 0.9pJ. However, reading a 32-bit operand from a register file is 5pJ, and reading that same word from external memory is 640pJ. This discrepancy of energy consumption between operator and data movement worsens with smaller manufacturing geometries, adding additional importance to maximize information content in arithmetic types to make them as dense and small as possible.

The performance of arithmetic types is also strongly impacted by data movement. Any memory-bound algorithm will benefit from moving fewer bits to and from external memory. Therefore, arithmetic types and algorithms must be co-designed to minimize the number of bits per operation to maximize performance.

For applications constrained by accuracy, such as simulation and optimization, arithmetic types need to cover the precision and dynamic range of the computation. The precision of a type is the difference between successive values representable by its encoding. The dynamic range is the difference between the smallest and the largest value representable by the encoding.

The domain of reliable numerical computing must provide verified answers. Arithmetic types for reliable computing need to guarantee numerical properties of the result [4]. For example, interval arithmetic can assert that the true computation answer lies in some interval and is an example of reliable computing.

Reproducible computation guarantees that results are the same regardless of the execution order. Reproducible computing is particularly pertinent for high concurrency environments, such as GPUs and High-Performance Computing (HPC) clusters.

3.3 Fixed Size, Arbitrary Precision

Fixed-size, arbitrary precision arithmetic types are tailored to energy-efficiency and memory bandwidth optimization.

integer<nbits, BlockType, NumberType> The **integer** arithmetic type can represent Natural Numbers, Whole Numbers, and integers of *nbits*. Natural and Whole Numbers are encoded as 1's complement numbers, and the Integers are encoded as a 2's complement numbers. Figure 2 shows a 16-bit incarnation. Its closure semantics are modulo, and effectively extend the C++ language with arbitrary precision signed integers of arbitrary fixed-size. This type is very common in hardware designs.

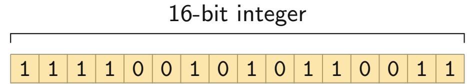

Fig. 2. A 16-bit integer.

fixpnt<nbits, rbits, arithmetic, BlockType> The **fixpnt** arithmetic type is a 2's complement encoded fixed-point of *nbits* with the radix point set at bit *rbits*. Figure 3 shows a 16-bit incarnation with the radix point at bit 8. Its closure semantics are configurable: either modulo or saturating. The fixpnt is constructed with blocks of type **BlockType**, and a fixpnt value is aligned in memory on **BlockType** boundaries.

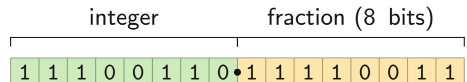

Fig. 3. A 16-bit fixed-point with 8 fraction bits.

cfloat<nbits, es, BlockType, sub, super, saturating> The **cfloat** arithmetic type is a floating-point type of size *nbits* bits, with 1 sign bit, an exponent field of *es* bits, and $nbits - 1 - es$ number of mantissa bits. The exponent is encoded as a biased integer. The mantissa is encoded with a hidden bit for normal and supernormals numbers. Subnormal numbers are numbers with all exponent bits set to 0s. Supernormal numbers are numbers with all exponent bits set to 1s. Normal numbers are all encodings that are not subnormal or supernormal. Closure semantics can be saturating or clipping to

$\pm\infty$. Figure 4 shows a 16-bit **cfloat** with 5 bits of exponent. When subnormals are selected and supernormals and saturating are deselected, this would represent a half-precision IEEE-754 FP16. If subnormals are deselected, it will represent the NVIDIA and AMD FP16 arithmetic type. Figure 5 shows a single-precision floating-point. The **cfloat** type can represent floating-point types ranging from 3 bits to thousands of bits, with or without subnormals, with or without supernormals, and with clipping or saturating closure semantics.

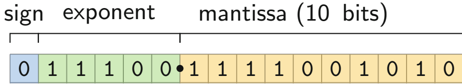

Fig. 4. Half-precision 16-bit floating-point representation, fp16.

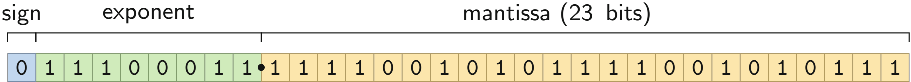

Fig. 5. Single precision 32-bit floating-point representation, fp32.

posit<nbits, es, BlockType> The **posit** arithmetic type represents a tapered floating-point type using the posit encoding. It offers a parameterized size of *nbits*, with 1 sign bit, *es* exponent bits, and $nbits - 3 - es$ mantissa bits around 1.0. Figure 6 shows a 16-bit posit with 3 exponent bits in regime -3.

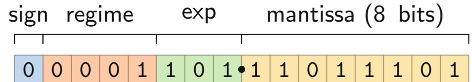

Fig. 6. A `posit<16,3>`.

lns<nbits, base, BlockType> The **lns** arithmetic type implements a logarithmic number system of size *nbits*, with *base* as the base.

The type set **integer**, **fixpnt**, **cfloat**, **posit**, and **lns** provide a productive baseline of arithmetic types that most developers are familiar with. *Universal* contains other types as well, including faithful types with uncertainty bits, type I and II **unum**s, and interval **posit**s, called **valid**s. These more advanced arithmetic types provide facilities for reliable computing and numerical analysis.

3.4 Variable Size, Adaptive Precision

Adaptive precision arithmetic types are tailored to questions regarding numerical precision and computational accuracy. This has been the traditional domain of numerical research platforms, such as `Boost Multiprecision`, `MPFR`, and `GMP` [5,7,19]. *Universal* currently offers only two adaptive precision arithmetic types:

- decimal
- rational

Implementation work has started on adaptive precision floating-point based on Douglas Priest's work [24], and the lazy exact arithmetic type proposed by Ryan McCleeary [20].

4 Creating a New Arithmetic Type

The experience with implementing a dozen or so arithmetic types has exposed a typical pattern of how to quickly and reliably bring up a new arithmetic type. The first step is to define the memory layout of the parameterized type. This step blocks the storage required to contain the encoded bits. *Universal* exposed the block type used for storage and alignment. For example, using a `uint8_t` as the building block, the memory layout of the individual value would be the minimum number of blocks to contain the encoding. The memory alignment would be on byte boundaries.

Once the memory layout has been designed, the next step is to implement the conversion from encoding to native types, such as **float** or **double**. Provide a simple set of `convert_to_` methods to test the interpretation of bits in the encoding to generate tables to validate the encoding.

The next step is to implement the inverse transformation - the conversion from native types to the encoding of the new arithmetic type. This conversion tends to be the most involved algorithmic task as sampling the native type values by the new arithmetic type requires robust rounding decisions. The conversion regression suite of this step is also involved as one needs to enumerate all possible rounding situations across all possible encodings.

Once we have the memory layout, encoding, and the two conversion directions, the type can be used for computation by simply converting the value to a native type, calling arithmetic operators or math library functions, and converting the result back into the encoding.

The final two implementation tasks are native arithmetic operators and native implementations of the elementary functions. Native implementations are the only safeguard against double-conversion errors. Native implementations are also crucial for performance and hardware validation. The arithmetic operators vary by which algebraic system the arithmetic type is associated. Still, in general, we need to implement the following set:

- addition
- subtraction

- multiplication
- division
- remainder
- square root

The final step in creating a new arithmetic type is to provide native implementations of the elementary functions. To do this for parameterized types is still an open research question, as the approximation polynomials, albeit *minimax* or *Minefield*, are specific to each configuration.

5 Arithmetic Type API

5.1 Construction

All Universal arithmetic types have a default, copy, and move constructor. This allows the application to create, copy, and efficiently call return values.

```
// required constructors
constexpr posit() noexcept
constexpr posit(const posit&) noexcept
constexpr posit(posit&&) noexcept
```

To support efficient conversions between native types and the user defined type, we encourage to provide converting constructors for all native types. Casting to the largest native type would create inefficiency specifically for small encodings where performance is most desired.

```
// signed native integer types
constexpr posit(signed char) noexcept
constexpr posit(short) noexcept
constexpr posit(int) noexcept
constexpr posit(long) noexcept
constexpr posit(long long) noexcept

// unsigned native integer types
constexpr posit(char) noexcept
constexpr posit(unsigned short) noexcept
constexpr posit(unsigned int) noexcept
constexpr posit(unsigned long) noexcept
constexpr posit(unsigned long long) noexcept

// native floating-point types
constexpr posit(float) noexcept
constexpr posit(double) noexcept
constexpr posit(long double) noexcept
```

Some compilers, `Clang` in particular, treat type aliases as different types. Aliases such as `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` are not equivalent to `char`, `unsigned short`, `unsigned int`, and `unsigned long long`, respectively. This causes potential compilation problems when using type aliases in converting constructors. Instead of matching the appropriate constructor, your code will go through an implicit conversion, which can cause latent bugs that are hard to find. Best is to specialize on the native language types, `short`, `int`, `long`, etc.

5.2 Assignment

Assignment operators follow the same structure as the converting constructors.

```
// assignment operators for native types

// signed native integer types
constexpr fixpnt& operator=(signed char rhs)        noexcept
constexpr fixpnt& operator=(short rhs)              noexcept
constexpr fixpnt& operator=(int rhs)                noexcept
constexpr fixpnt& operator=(long rhs)               noexcept
constexpr fixpnt& operator=(long long rhs)          noexcept

// unsigned native integer types
constexpr fixpnt& operator=(char rhs)               noexcept
constexpr fixpnt& operator=(unsigned short rhs)     noexcept
constexpr fixpnt& operator=(unsigned int rhs)       noexcept
constexpr fixpnt& operator=(unsigned long rhs)      noexcept
constexpr fixpnt& operator=(unsigned long long rhs) noexcept

// native floating-point types
constexpr fixpnt& operator=(float rhs)              noexcept
constexpr fixpnt& operator=(double rhs)             noexcept
```

Another compiler environment constraint, particularly for embedded environments, is support for long double. Embedded ARM and RISC-V compiler environments do not support long double, so Universal guards the long double construction/conversion and must be explicitly enabled.

```
// guard long double support to enable
// ARM and RISC-V embedded environments
#if LONG_DOUBLE_SUPPORT
constexpr fixpnt(long double initial_value)         noexcept
constexpr fixpnt& operator=(long double rhs)        noexcept
constexpr explicit operator long double() const     noexcept
#endif
```

5.3 Conversion

Operators that convert from native types to the custom type are provided through converting constructors and assignment operators. However, the conversion from custom type to native types is marked `explicit` to avoid implicit conversions that can hide rounding errors that are impossible to isolate.

```
// make conversions to native types explicit
explicit operator int()         const
explicit operator long long()   const
explicit operator double()      const
explicit operator float()       const
explicit operator long double() const
```

5.4 Logic Operators

The *Universal* arithmetic types are designed to be plug-in replacements for native types. Notably, for the logic operators in the language, it is common to come across this code:

```
...
cfloat<16,5> a, b, c;
...
if (b != 0) c = a / b;
...
```

Porting existing codes to use *Universal* types provided evidence that all and every combination of literal comparisons are used. The logic operator design must thus be complete and capture all combinations of arithmetic type and literal type that are native to the language.

The design in *Universal* uses a strongly typed set of operator signatures that provide an optimized implementation for the comparison leveraging the size of the literal. Typically, we only need to implement `operator==()` and `operator<()` with native encoding knowledge. The other logic operators can be expressed in terms of these two operators. The exception to this rule is the IEEE-754 derived arithmetic types with NaN encodings. In those systems, the logic operators are not complementary, and each operator requires decision code to deal with this particular type.

```
// base logic operators are defined as friends
template<size_t nbits, size_t es>
friend bool operator==(const valid<nbits, es>& lhs,
                       const valid<nbits, es>& rhs);
template<size_t nbits, size_t es>
friend bool operator!=(const valid<nbits, es>& lhs,
                       const valid<nbits, es>& rhs);
template<size_t nbits, size_t es>
friend bool operator< (const valid<nbits, es>& lhs,
                       const valid<nbits, es>& rhs);
template<size_t nbits, size_t es>
friend bool operator> (const valid<nbits, es>& lhs,
                       const valid<nbits, es>& rhs);
template<size_t nbits, size_t es>
friend bool operator<=(const valid<nbits, es>& lhs,
                       const valid<nbits, es>& rhs);
template<size_t nbits, size_t es>
friend bool operator>=(const valid<nbits, es>& lhs,
                       const valid<nbits, es>& rhs);
```

There are three signed integers (`int`, `long` and `long long`), three unsigned, and three floating-point (`float`, `double`, and `long double`) literals. This creates the need for $6 \times 3 \times 3 \times 3 \times 2 = 324$ free functions to capture all the combinations between arithmetic type and literal. These free functions will transform the literal to the arithmetic type and then call the operational layer to execute the comparison. Judicious use of implicit conversion rules can be used to reduce that number, but care must be taken to avoid double rounding errors.

5.5 Arithmetic Operators

Binary arithmetic operators are implemented through free binary functions that capture literals and type conversions coupled with in-place update class operators.

```
// update operators
cfloat& operator+=(const cfloat& rhs)
cfloat& operator-=(const cfloat& rhs)
cfloat& operator*=(const cfloat& rhs)
cfloat& operator/=(const cfloat& rhs)

// free binary functions to transform literals
template<size_t nbits, size_t es, typename bt,
bool hasSubnormals, bool hasSupernormals, bool isSaturating>
cfloat<nbits, es, bt,
       hasSubnormals, hasSupernormals, isSaturating>
operator+(const double lhs,
          const cfloat<nbits, es, bt,
                       hasSubnormals, hasSupernormals,
                       isSaturating>& rhs) {
        cfloat<nbits, es, bt, hasSubnormals, hasSupernormals,
               isSaturating> sum(lhs);
        sum += rhs;
        return sum;
}
```

5.6 Serialization

All arithmetic types in *Universal* support serialization through the stream libraries.

```
std::ostream& operator<<(std::ostream& ostr,
                          const decimal& d)
std::istream& operator>>(std::istream& istr, decimal& p)
```

Such conversions may introduce rounding errors, so *Universal* types also support a error free **ASCII** format. This is controlled by a compilation guard:

```
///////////////
// enable/disable special posit format I/O
#if !defined(POSIT_ERROR_FREE_IO_FORMAT)
// default is to print double values
#define POSIT_ERROR_FREE_IO_FORMAT 0
#endif
```

When error-free printing is enabled, values are printed with a designation and a hex format to represent the bits. Here is an example of a posit use case:

```
posit<32,2> p(1.0);
cout << "Error free posit value: " << p << endl;
...
> Error free posit value: 32.2x40000000p
```

5.7 Set and Query Interface

To support efficient verification and validation of an arithmetic type, the regression suites need to be able to set and query bits.

```
// modifiers
inline constexpr void clear() noexcept
inline constexpr void setzero() noexcept
inline constexpr void setbit(size_t i, bool v = true)
inline constexpr void setbits(uint64_t raw_bits) noexcept
inline constexpr void setblock(size_t b, const BlockType& data) noexcept
```

The methods `setbit()` and `setbits()` make it possible to write generic regression tests for arithmetic types, thus drastically reducing the amount of code that needs to be written to validate the arithmetic types in *Universal*. This basic API can be augmented to set special encodings for specific arithmetic types. For example, here is the extended set for *cfloat*:

```
inline constexpr void setinf(bool sign = true) noexcept
inline constexpr void setnan(int NaNType = NAN_TYPE_SIGNALLING) noexcept
inline constexpr void setsign(bool sign = true)
inline constexpr bool setexponent(int scale)
inline constexpr void setfraction(uint64_t raw_bits)
```

The verification phase is aided by a productive reflection interface. The *Universal* arithmetic types have a standard set of state query methods to simplify the verification algorithms.

```
// selectors
inline constexpr bool sign() const noexcept
inline constexpr int  scale() const noexcept
inline constexpr bool ispos() const noexcept
inline constexpr bool iszero() const noexcept
inline constexpr bool isone() const noexcept
```

Just like the modifiers, the basic API can be augmented to capture specific state: here is the extended set for *cfloat*.

```
// special value queries for cfloat
inline constexpr bool isinf(int InfType) const noexcept
inline constexpr bool isnan(int NaNType) const noexcept

// range queries for cfloat
inline constexpr bool isnormal() const noexcept
inline constexpr bool isdenormal() const noexcept
inline constexpr bool issupernormal() const noexcept
```

5.8 Support Functions and Manipulators

Working with bit encodings is challenging, so *Universal* provides a collection of manipulators and support functions to ease queries and interpret bit encodings. The manipulator `color_print` color-codes different segments of encoding so that it is easier to decipher and compare, as is shown in Fig. 7 for different posit configurations.

```
posit< 8,0>  = 01101001 : 3.125
posit< 8,1>  = 01011001 : 3.125
posit< 8,2>  = 01001101 : 3.25
posit< 8,3>  = 01000110 : 3
posit<16,1>  = 0101100100100010 : 3.1416
posit<16,2>  = 0100110010010001 : 3.1416
posit<16,3>  = 0100011001001000 : 3.1406
posit<24,1>  = 010110010010000111111011 : 3.141592
posit<24,2>  = 010011001001000011111110 : 3.141594
posit<24,3>  = 010001100100100001111111 : 3.141594
posit<32,1>  = 01011001001000011111101101010100 : 3.14159265
posit<32,2>  = 01001100100100001111110110101010 : 3.14159265
posit<32,3>  = 01000110010010000111111011010101 : 3.14159265
posit<48,1>  = 010110010010000111111011010101000100010000101101 : 3.1415926535898
posit<48,2>  = 010011001001000011111101101010100010001000010111 : 3.1415926535899
posit<48,3>  = 010001100100100001111110110101010001000100001011 : 3.1415926535897
posit<64,1>  = 0101100100100001111110110101010001000100001011010001100000000000 : 3.1415926535897931 2
posit<64,2>  = 0100110010010000111111011010101000100010000101101000110000000000 : 3.1415926535897931 2
posit<64,3>  = 0100011001001000011111101101010100010001000010110100011000000000 : 3.1415926535897931 2
posit<64,4>  = 0100001100100100001111110110101010001000100001011010001100000000 : 3.1415926535897931 2
```

Fig. 7. Demonstration of the `color_print(p)` function.

6 Example: Matrix Scaling and Equilibrating

Several multi-precision iterative-based algorithms for solving $Ax = b$ have been developed [2,3,8,10] showing speedup over double precision solvers (e.g., [11]). In those studies, the algorithms round the entries of A to lower precision (i.e., half precision), perform LU decomposition, compute a solution using the low accuracy LU factors, then use iterative refinement back to working precision. The results are highly dependent on the condition number of the matrix. It is worth noting that even well-conditioned matrices can become ill-conditioned in lower precision. Because when scaling to lower precision, underflow may produce a singular matrix, the percentage of nonzero elements after scaling to lower precision is critical. Rounding to lower precision also can result in overflow (or subnormals); however, overflow is far less likely in scientific computing applications [12]. Next, we outline three algorithms (see Appendix A) presented in [12] using slightly different notation.

Let p represent the precision (e.g., `fp16`), $a_{\max} = \max_{i,j} |a_{i,j}|$, and $x_{\max}$ the largest positive value represented in the lower precision arithmetic. The goal of scaling is to reduce the condition number and to increase speed in solving. Table 1 lists key characteristics of half precision, double precision, and `posit<16,2>`.

Algorithm 1 converts all entries to lower precision format. Any entry that rounds to infinity is replaced by the maximum signed value. However, Algorithm 1 does not handle underflow or subnormal situations. Furthermore, it alters the matrix significantly when $|a_{ij}| \gg x_{\max}$, (see [12]). Algorithm 2 on the other hand, scales then rounds in such a way to avoid overflow, however shrinks the magnitude of each increasing the chance of underflow, thus increasing the possibility of producing a singular matrix. Algorithm 3 addresses those issues by equilibrating rows and columns so that the maximum entry is 1 in each row and column. We note it is not possible to underflow using posit configurations, so there will be no reduction in the number nonzero elements. As such, more research on how and when to scale is needed. Universal permits testing multiple configurations and provides a vehicle for this research.

Table 1. Specifications of multiple binary formats. Precision of the arithmetic is measured by the unit round off, which is 2^{-p}, where p is the number of fraction bits.

Format	Fraction	Exponent	ulp	Min	Max
posit<16,2>	$\leq$ 11	2	4.88e−04	1.39e−17	7.21e+16
fp16	11	5	4.88e−04	6.10e−05	6.55e+04
fp64	53	11	1.11e−16	2.22e−308	1.80e+308

7 Conclusions and Future Work

We have presented the current status of the *Universal* library `v3`. The third edition of the library contains the core arithmetic types that cover the algebraic sets and contains a unified API to simplify validation and regression testing. We provided some examples of the arithmetic types' intended use and reported on some framework integrations for research and development of computational science and engineering applications. Much work remains. The Oracle-style arithmetic types need to be fleshed out so that the mathematical library work can move forward. Math libraries that are parameterized with arithmetic type attributes such as precision and dynamic range are still an open research question and, when resolved, would dramatically improve the code-efficiency of *Universal*. Many new arithmetic types are being proposed for Deep Learning and Optimal control that need representations that are more energy-efficient. Moreover, many next-generation application platforms are written in Python, Julia, Golang, and Rust, and will need integration facilities to leverage the arithmetic types in *Universal*. The current incarnation of the library provides productive facilities to research and develop energy-efficient and high-performance algorithms through custom arithmetic. The *Universal* library development is managed as an open-source community project [1] where all contributions are welcome.

Appendix A: Squeezing Algorithms

Algorithm 1: Round and replace overflow with $x_{\max}$.

Input: An $n \times n$ Matrix A
Output: Rounded matrix B

1 $B = \mathrm{fl_p(A)}$
2 Set $a_{ij}^{p} = \mathrm{sign(a_{ij})x_{max}}$

Algorithm 2: Scaled matrix entries

Input: An $n \times n$ Matrix A
Output: Rounded matrix $A^{(p)}$

1 $a_{\max} = \max_{i,j} |a_{i,j}|$
2 $\mu = x_{\max}/a_{\max}$
3 $A^{(p)} = \mathrm{fl_p}(\mu \mathrm{A})$

Algorithm 3: Double side scaling using row and column equilibration

Input: An $n \times n$ Matrix A
Output: Rounded matrix $A^{(p)}$

1 Set $R = 0$
2 **for** $i = 1$ **to** n **do**
3 | $R(i,i) \leftarrow ||A(i,:)||_\infty^{-1}$
4 **end**
5 $B = RA$
6 Set $S = 0$
7 **for** $j = 1$ **to** n **do**
8 | $S(j,j) \leftarrow ||A(:,j)||_\infty^{-1}$
9 **end**
10 Set $\beta =$ maximum absolute entry in RAS
11 Set $\mu = x_{\max}/\beta$
12 Return $\mathrm{fl_p}(\mu(\mathrm{RAS})$

References

1. Universal number library (2017). https://github.com/stillwater-sc/universal
2. Carson, E., Higham, N.J.: A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. SIAM J. Sci. Comput. **39**(6), A2834–A2856 (2017)
3. Carson, E., Higham, N.J.: Accelerating the solution of linear systems by iterative refinement in three precisions. SIAM J. Sci. Comput. **40**(2), A817–A847 (2018)
4. Cox, M.G., Hammarling, S.: Reliable Numerical Computation. Clarendon Press, Oxford (1990)
5. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: a multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. (TOMS) **33**(2), 13-es (2007)
6. Gottschling, P., Wise, D.S., Adams, M.D.: Representation-transparent matrix algorithms with scalable performance. In: Proceedings of the 21st Annual International Conference on Supercomputing, pp. 116–125 (2007)
7. Granlund, T.: GNU MP. The GNU Multiple Precision Arithmetic Library **2**(2) (1996)
8. Gupta, S., Agrawal, A., Gopalakrishnan, K., Narayanan, P.: Deep learning with limited numerical precision. In: International Conference on Machine Learning, pp. 1737–1746. PMLR (2015)
9. Gustafson, J.L., Yonemoto, I.T.: Beating floating point at its own game: posit arithmetic. Supercomput. Frontiers Innovations **4**(2), 71–86 (2017)

10. Haidar, A., Tomov, S., Dongarra, J., Higham, N.J.: Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 603–613. IEEE (2018)
11. Haidar, A., Wu, P., Tomov, S., Dongarra, J.: Investigating half precision arithmetic to accelerate dense linear system solvers. In: Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, pp. 1–8 (2017)
12. Higham, N.J., Pranesh, S., Zounon, M.: Squeezing a matrix into half precision, with an application to solving linear systems. SIAM J. Sci. Comput. **41**(4), A2536–A2551 (2019)
13. Hittinger, J., et al.: Variable precision computing. Technical report, Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States) (2019)
14. Horowitz, M.: 1.1 computing's energy problem (and what we can do about it). In: 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), pp. 10–14. IEEE (2014)
15. Intel Corporation: BFLOAT16 - Hardware Numerics Definition (2018). https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf
16. Jouppi, N.P., et al.: In-Datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, pp. 1–12 (2017)
17. Kharya, P.: TensorFloat-32 in the a100 GPU accelerates AI training HPC up to 20x. NVIDIA Corporation, Technical report (2020). https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/
18. Lloyd, G.S., Lindstrom, P.G.: ZFP hardware implementation. Technical report, Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States) (2020)
19. Maddock, J., Kormanyos, C., et al.: Boost multiprecision (2018)
20. McCleeary, R.: Lazy exact real arithmetic using floating point operations (2019)
21. Molisch, A.F., et al.: Hybrid beamforming for massive MIMO: a survey. IEEE Commun. Mag. **55**(9), 134–141 (2017)
22. Muhammad, K., Ullah, A., Lloret, J., Del Ser, J., de Albuquerque, V.H.C.: Deep learning for safe autonomous driving: current challenges and future directions. IEEE Trans. Intell. Transp. Syst. **22**(7), 4316–4336 (2020)
23. Omtzigt, E.T.L., Gottschling, P., Seligman, M., Zorn, W.: Universal numbers library: design and implementation of a high-performance reproducible number systems library. arXiv:2012.11011 (2020). https://arxiv.org/abs/2012.11011
24. Priest, D.M.: Algorithms for Arbitrary Precision Floating Point Arithmetic. University of California, Berkeley (1991)
25. Siek, J.G., Lumsdaine, A.: The matrix template library: a generic programming approach to high performance numerical linear algebra. In: Caromel, D., Oldehoeft, R.R., Tholburn, M. (eds.) ISCOPE 1998. LNCS, vol. 1505, pp. 59–70. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49372-7_6
26. Siek, J.G., Lumsdaine, A.: The matrix template library: a unifying framework for numerical linear algebra. In: Demeyer, S., Bosch, J. (eds.) ECOOP 1998. LNCS, vol. 1543, pp. 466–467. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49255-0_152
27. Dally, W.J., et al.: Neural network accelerator using logarithmic-based arithmetic (2021). https://uspto.report/patent/app/20210056397