Safe Arithmetic User Guide

Table of Contents

1. Motivation

C++ represents numerical values with finite bounds and discrete steps, as do all commonly used programming languages and processors. This is not a problem as long as specific rules regarding overflow, underflow, and loss of precision are meticulously followed. There are a number of guidelines and standards published for functional safety and security that detail the rules for safe arithmetic operations as well as many other concerns:

- MISRA C and MISRA C++ Guidelines
- CERT C and CERT C++ Secure Coding Standards
- ISO/IEC TS 17961:2013 (C Secure Coding Rules)
- IEC 61508 (Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems)

These publications represent a large investment in the safe application of software, and for good reason. The consequences for such bugs can be catastrophic.

- Therac-25 radiotherapy machine. An integer overflow caused the machine to bypass software safety checks and deliver deadly radiotherapy doses to patients.
- Ariane rocket flight V88. An integer overflow was caught, but lead to an exception that was handled incorrectly, thus halting its inertial navigation system. This ultimately led to the catastrophic destruction of the rocket and its payload.
- Boeing 787 computer crash. Two different integer overflows caused either misleading data to be displayed or systems crash.

There are countless other functional bugs and security vulnerabilities all due to unsafe arithmetic operations.

The Safe Arithmetic library is intended and designed to protect against unsafe arithmetic operations. This includes overflow, underflow, and divide-by-zero. It follows the "Valley of success" strategy; The natural state of a software project utilizing the Safe Arithmetic library contains clear boundaries between safe and unsafe arithmetic. Within the safe arithmetic environment the easiest and simplest thing to do is also the safest thing to do. In this context, arithmetic includes assignment and construction.

1.1. Numeric Type Support

The current version of the Safe Arithmetic library supports the following value types:

- C++20 integer primitives
- `safe::integer<N>` arbitrary precision integers

In the future, it is likely support for the following numeric types will be added:

- Fixed point
- Pointers

It is unlikely that other numeric types, like `float` and `double` will be supported in the near-term. They at least are not needed by the original author of the library. That being said, pull-requests are more than welcome and we greatly appreciate community involvement.

1.2. Library Inter-op

1.2.1. Dimensional Analysis

Dimensional analysis libraries in C++ typically use the language's type system to enforce correct assignment of values with dimensions to variables or function arguments and forbid nonsensical operations like adding a length value to a temperature value. Multiplication and division operations of differing dimensions are allowed and the resulting type represents a new unit and dimension.

The Safe Arithmetic library also uses the C++ type system to enforce correct operations by encoding the set of possible values in a `safe::var` 's type. Operations on these `safe::var` s return new types with an updated set of possible values. The actual runtime value is guaranteed to be within this set.

The C++ type system essentially supports sophisticated static analysis through compilation of the code. It is additionally possible to leverage a dimensional analysis library like `mp-units` as well as the Safe Arithmetic library to further increase confidence in your code.

1.2.2. Messaging

Intel's Compile-time Initialization and Build library's messaging capabilities may also be used with great success with the Safe Arithmetic library. Message fields may be given safe types to enforce correct assignment to fields as well as provide field values that are already wrapped in safe types when read.

1.2.3. Register Access

Intel's internal Config Register Operation Optimizer library may be used with the Safe Arithmetic library to ensure safe and correct access of registers. Without the Safe Arithmetic library, it is possible to write values too large to be represented by a register field's bit width. With the Safe Arithmetic library, it is possible to ensure only values in range are written to fields and field read types will also be automatically wrapped in an appropriately constrained safe type.

1.3. Similar Libraries

Safe Arithmetic is not the only library that attempts to solve the problem of unsafe integer or numeric operations in programs. There are a number of other libraries with different strategies and tradeoffs that are worthwhile to look at.

- Boost's safe_numerics. A library by Robert Ramey included in Boost. It works on the C++14 standard rather than C++20 as required by Safe Arithmetic. It offers more customization points on how exceptional cases are handled: compile-time, runtime exceptions, or a custom error handler. It also allows customization for how integer promotion is handled. It has been around longer than Safe Arithmetic. It only supports intervals to specify value requirements, while Safe Arithmetic implements a DSL to allow more tightly-constrained requirements.
- bounded::integer. A library by David Stone. Defines a `bounded::integer<Min, Max>` template type that tracks the set of potential values of operations using interval math. Implemented in C++20 as modules.
- SafeInt. Provides both a C++11 and C library implementation. Appears to only support runtime checking.

- PSsimplesafeint. "A C++20 implementation of safe (wrap around) integers following MISRA C++ rules."

2. Overview

2.1. Safe Arithmetic Environment

The Safe Arithmetic library uses the C++ type system to encode and enforce requirements on values. A special template type, `safe::var` is used to contain these values.

```
namespaced safe {
    template<typename ValueType, auto Requirements>
    struct var;
}
```

Arithmetic, bitwise, and shift operators on `safe::var` values results in the generation of a new `safe::var` with its requirements updated to represent the set of possible values the result may contain. Operations on `safe::var` values are guaranteed to be safe at compile-time. There is no runtime overhead incurred. Only the desired operations are performed.

Operations on instances of `safe::var` forms a hermetically sealed context in which overflows, underflows, and division-by-zero are proven impossible by the Safe Arithmetic library implementation. If such a condition were possible due to an arithmetic operation on a `safe::var`, then compilation would fail.

This leaves two important questions, how to get values in and out of this "Safe Arithmetic Environment".

2.2. Creating Safe Values

Safe literal values can be created using the `_i` user defined literal. It will create a `safe::var` with the necessary integer type to contain the value and a requirement that matches the value. Literal values larger than 64-bits are implemented using an arbitrary precision integer type built into the Safe Arithmetic library.

```
namespace safe::literals {
    template <char... chars>
    constexpr auto operator""_i();
}
```

Safe versions of the C++ primitive integer types are available for declaring runtime values. Each primitive integer type is wrapped in `safe::var` with a requirement describing the range of that primitive type.

```
namespace safe {
```

```
    // safe versions of C++ primitive integer types
    using u8 = var<std::uint8_t, ...>;
    using s8 = var<std::int8_t, ...>;
    using u16 = var<std::uint16_t, ...>;
    using s16 = var<std::int16_t, ...>;
    using u32 = var<std::uint32_t, ...>;
    using s32 = var<std::int32_t, ...>;
    using u64 = var<std::uint64_t, ...>;
    using s64 = var<std::int64_t, ...>;
}
```

Because each safe primitive integer type requirement's can contain all values that are representable by the underlying type, it is safe to directly assign primitive values to instances of these safe primitive types.

```
// SAFE: The signed literal value is gauranteed to fit in a std::int64_t.
safe::s64 foo = 0xc001bea7;

// COMPILE ERROR: On systems with 32- or 64-bit int types, the literal will be
// too large to fit and compilation will fail.
safe::s16 bar_1 = 0xba11;

// SAFE: The safe integer UDL encodes its requirements in its type, the value
// is known at compile-time to fit in bar_2.
safe::s16 bar_2 = 0x5a5_i;

// SAFE: A safe 64-bit int can represent all values in an unsigned safe 16-bit int.
foo = bar_1;

// SAFE: A safe 32-bit unsigned int can safely be assigned the value of a uint32_t
volatile std::uint32_t my_fav_reg;
safe::u32 safe_reg_val = my_fav_reg;

// SAFE: Masking safe_reg_val by an 8-bit value guarantees the result will fit
// in a safe::u8
safe::u8 safe_reg_field_val_1 = safe_reg_val & 0xFF_i;

// COMPILE ERROR: safe::u8 cannot represent all possible values of these rhs
// variables.
safe::u8 safe_reg_field_val_2 = safe_reg_val;
safe::u8 safe_reg_field_val_3 = my_fav_reg;
```

For integer values that cannot fit in the primitive types provided by C++, the Safe Arithmetic library provides an arbitrary precision implementation, `safe::integer`.

```
namespace safe {
    // safe arbitrary precision signed integer type
    template<std::size_t NumBits>
```

```
    using integer = var<...>;
}
```

Safe Arithmetic's integer promotion rules will automatically pick an integer type large enough to represent the possible values of an arithmetic operation. There is little need to explicitly use `safe::integer`.

```
auto big_int_value = 0xba5eball'd00d5a7e'a11f00d5'900dg00b_i;

// SAFE: The result of the operation is known at compile time to fit.
safe::u64 small_1 = big_int_value >> 64_i;

// COMPILE ERROR: The result cannot be represented by a uint64_t.
safe::u64 small_2 = big_int_value >> 32_i;

volatile safe::u64 hw_reg_1;
volatile safe::u64 hw_reg_2;

// SAFE: The result type is automatically promoted to a safe::integer<66>. The
// 66th bit is the two's complement sign bit and the 65th bit is the carry from
// the 64-bit addition.
auto reg_result = hw_reg_1 + hw_reg_2;

// COMPILE ERROR: The addition of these unchecked values could overflow a
// safe:u64.
safe::u64 reg_result_2 = hw_reg_1 + hw_reg2;

// SAFE: The result is explicitly being masked to 64-bits.
safe::u64 reg_result_3 = (hw_reg_1 + hw_reg2) & 0xFFFF'FFFF'FFFF'FFFF_i;
```

`safe::match` is the only mechanism in the Safe Arithmetic library that will produce additional runtime overhead. It creates a callable object that may be called with `safe::var` or naked integer values. It uses compile-time checks if possible to match the given arguments with the `matchable_funcs` arguments. If compile-time checks are not possible for an argument, then the value is checked at runtime to determine if it satisfies the requirements for one of the `matchable_funcs`. It is analogous to a pattern matching switch statement where the `matchable_funcs` arguments `safe::var` requirements are the patterns to match the callable object's input arguments against.

This is the recommended way of marshalling external integer values into a safe arithmetic environment when the valid values are a subset of the underlying integer type's range. For example, an external value arrives in a full 16-bit unsigned integer, but the valid values are only 0 through 50,000. The full 16 bits are needed to represent the value, but only a subset of the 16-bit integer range is valid.

```
// Pseudo C++20 to illustrate safe::match API
template <typename F, typename Return, std::size_t NumArgs>
concept Callable = requires(F f, auto... args) {
    Return retval = f(args...);
```

```
    requires sizeof...(args) == NumArgs;
};

namespace safe {
    template<typename Return, std::size_t NumArgs>
    Callable<Return, NumArgs> auto match<Return>(
        Callable<Return, NumArgs> auto... matchable_funcs,
        Callable<Return, 0> auto default_func
    );
}
```

The operation of `safe::match` is easier to understand with some examples.

```
// Hardware register reporting a count of some event type.
volatile std::uint32_t event_counter_hw_reg;

// Hardware register representing the event type being reported.
volatile std::uint32_t event_type_hw_reg;

// Firmware array keeping track of updated event counts.
safe::array<safe::u16, 17> event_counts{};

constexpr auto process_event_count = safe::match<void>(
    [](
        safe::ival_u32<0, 1023> event_count,
        safe::ival_u32<0, 16> event_type
    ){
        auto const prev_count = event_counts[event_type];
        auto const new_count = prev_count + event_count;

        // this example is making the implementation choice of saturating the
        // event count to prevent overflow and rollover.
        event_counts[event_type] = max(new_count, safe::u16::max_value);
    },

    // Multiple functions with different requirements for parameters may be
    // passed in. The first function whose argument requirements are satisified
    // by the runtime argument values is executed. The last function must be
    // the default handler and is only executed if no prior match is found.

    [](){
        // default action, handle error condition as desired
    }
);

// Hardware triggers this interrupt every time a new event count is ready to
// be processed.
void event_count_interrupt_handler() {
    process_event_count(event_counter_hw_reg, event_type_hw_reg);
```

```
}
```

`safe::match` is a powerful tool that is discussed in more detail in the reference section.

The final method of introducing values into the safe arithmetic environment is through `unsafe_cast<T>(value)`. It bypasses all compile-time and runtime safety checks and depends on the value to be proven to satisfy the requirements of `T` using mechanisms outside the visibility and scope of the Safe Arithmetic library. Its use is highly discouraged. The name is chosen to cause an uneasy feeling in programmers and clearly signal a red flag for code reviewers.

```
template<typename T>
T unsafe_cast(auto value);
```

`unsafe_cast<T>(value)` is used within the Safe Arithmetic library to ferry values into instances of `safe::var` after proving it is safe to do so. It is necessary for the library's construction.

As always, an example is useful to illustrate how to use a particular function.

```
std::uint16_t some_function();
void do_something_useful(safe::ival_u32<0, 1024> useful_value);

// VERY DANGEROUS: Don't do this!
auto dangerous_value = unsafe_cast<safe::ival_u32<0, 1024>>(some_function());
do_something_useful(dangerous_value);

// SAFE: Use safe::match instead. This will only call 'do_something_useful'
// if the result of 'some_function' satisfies the requirements on
// 'useful_value'. If it doesn't match, the default callable will be invoked.
safe::match<void>(do_something_useful, [](){})(some_function());

// SAFE: Don't use unsafe_cast<T>(value), try almost everything else first.
```

If you find a case where you feel you must use `unsafe_cast`, then maybe there is a gap in the Safe Arithmetic API or an algorithm that is missing. Please let us know by filing an issue.

2.3. Extracting Safe Values

Extracting values out of the safe arithmetic environment is not dangerous or unsafe in itself, but it is important to be explicit when doing so. `safe_cast<T>(value)` is used to extract integer values out of `safe::vars`. It is an acknowledgement by the programmer they are leaving the safe environment and must now take on the burden of ensuring safe arithmetic operations manually. It is also a clear indication for code reviewers to take a more critical look at any following integer operations.

```
template<typename T>
T safe_cast(auto value);
```

```
safe::ival_s32<-1000, 1000> my_safe_value = 42_i;

// SAFE: It's OK to use safe_cast to assign to a wider primitive type
auto innocent_value = safe_cast<std::int32_t>(my_safe_value);

// COMPILE ERROR: A narrowing conversion is not allowed by safe_cast
auto another_innocent = safe_cast<std::int8_t>(my_safe_value);
```

2.4. Safe Arithmetic Requirements DSL

The Requirements Domain-Specific Language is used to define the set of valid values for a `safe::var<T, Requirements>` templated type. `safe_numerics` and `bounded::integer` both use interval arithmetic at compile time to track the set of valid values. The Safe Arithmetic library works with intervals, sets, tristate bitmasks, and set operators like union, intersection, and difference to define arbitrary requirements on values. Just like `safe_numerics` and `bounded::integer`, it will calculate the new set of possible values for any arithmetic, bitwise, or shift operation.

Since interval requirements are commonly used, there are convenience types for creating them:

```
safe::ival_s32<-100, 100> small_number{};
```

Which is equivalent to the following:

```
safe::var<std::int32_t, safe::ival<-100, 100>> small_number = 0_i;
```

If we want to exclude '0' from the range, the DSL allows us to do that:

```
using safe::ival;
safe::var<std::int32_t, ival<-100, -1> || ival<1, 100>> small_nonzero_number = 1_i;
```

This enables the library to protect against divide-by-zero at compile-time. The division operator function arguments require the divisor to be non-zero.

```
// COMPILE ERROR: small_number _might_ be zero
auto result_1 = 10_i / small_number;

// SAFE: small_nonzero_number is guaranteed to be non-zero.
auto result_2 = 10_i / small_nonzero_number;
```

The DSL can be used by itself, outside of `safe::var`. This can be helpful to illustrate the rules and capabilities of the DSL itself.

The assignment operator and constructors for `safe::var<T, Req>` that accept another `safe::var<RhsT, RhsReq>` use set inequality operators to determine whether it is safe or not. The

right-hand-side argument's requirements must be a subset of the left-hand-side target.

```
using safe::ival;

constexpr auto non_zero_req = ival<-100, -1> || ival<1, 100>;
constexpr auto small_num_req = ival<-100, 100>;

// The `<=` operator is used for 'is subset of'
static_assert(non_zero_req <= small_num_req);

safe::var<std:int32_t, non_zero_req> non_zero = 1_i;

// The `<=` operator ensures this assignment is safe at compile-time
safe::var<std::int32_t, small_num_req> small_num = non_zero;
```

When any operation is performed on a `safe::var` instance, the mirror operation is performed on the requirements.

```
using safe::ival;

constexpr auto one_to_ten_req = ival<1, 10>;
constexpr auto non_zero_req = ival<-100, -1> || ival<1, 100>;

safe::var<std:int32_t, non_zero_req> a = 42_i;
safe::var<std:int32_t, one_to_ten_req> b = 3_i;

auto c = a * b;

// runtime value is updated as expected
assert(c == 126_i);

// static requirements are also updated as expected
static_assert(c.requirement == ival<-1000, -1> || ival<1, 1000>);
```

2.4.1. Requirement DSL Primitives

Name	Definition	C++	Description
Interval	`[a, b]`	`safe::ival<a, b>`	A set of values from a to b, inclusive.
Set	`{a, b, c, ...}`	`safe::set<a, b, c, ⋯>`	A set of explicitly defined values.

Name	Definition	C++	Description
Mask	Let V be an integer and V_i be its i^{th} binary digit Let C be an integer and C_i be its i^{th} binary digit Let x_i be the i^{th} binary digit of x `{x in NN | 0 <= x < 2^n ^^ AAi (V_i vv (C_i = x_i))}`	`safe::mask<V, C>`	V is the variable bits mask. C is the constant bits mask. `safe::mask` produces a set of integers where the binary digits match C if the corresponding digits of V are unset. The binary digit places that are set in V are unconstrained in the elements of the produced set.

2.4.2. Requirement DSL Operators

Name	Definition	C++ Operator	Description
Subset	`A subseteq B`	<code><code>A <= B</code></code>	Test if A is a subset of B.
Superset	`A supe B`	`A >= B`	Test if A is a superset of B.
Set Equality	`A = B`	`A == B`	Test if A and B contain identical elements.
Set Inequality	`A != B`	`A != B`	Test if A and B do not contain identical elements.
Set Union	`A uu B`	`A || B`	Set of all elements in A and B.
Set Intersection	`A nn B`	`A && B`	Set of common elements in A and B.
Addition	`{a + b | a in A, b in B}`	`A + B`	Set of product pairs of A and B added.
Subtraction	`{a - b | a in A, b in B}`	`A - B`	Set of product pairs of A and B subtracted.
Multiplication	`{a * b | a in A, b in B}`	`A * B`	Set of product pairs of A and B multiplied.
Division	`{a / b | a in A, b in B}`	`A / B`	Set of product pairs of A and B divided.
Modulo	`{a % b | a in A, b in B}`	`A % B`	Set of product pairs of A and B modulo.

Name	Definition	C++ Operator	Description
Absolute Value	`{|a| | a in A}`	`abs(A)`	Set of the absolute value of all elements in A.
Minimum Value	`{min(a, b) | a in A, b in B}`	`min(A, B)`	Set of the minimum of each product pair of A and B.
Maximum Value	`{max(a, b) | a in A, b in B}`	`max(A, B)`	Set of the maximum of each product pair of A and B.
Bitwise AND	`{a\ &\ b | a in A, b in B}`	`A & B`	Set of product pairs of A and B bitwise ANDed.
Bitwise OR	`{a | b \ |\ a in A, b in B}`	`A | B`	Set of product pairs of A and B bitwise ORed.
Bitwise XOR	`{a\ o+\ b | a in A, b in B}`	`A ^ B`	Set of product pairs of A and B bitwise XORed.
Bitwise NOT	`{~a | a in A}`	`~A`	Bitwise NOT of all elements in A.
Bitwise Shift Left	`{a " << " b | a in A, b in B}`	`A << B`	Set of product pairs of A and B bitwise shifted left.
Bitwise Shift Right	`{a " >> " b | a in A, b in B}`	`A >> B`	Set of product pairs of A and B bitwise shifted right.

3. API Reference

3.1. safe::var

```
namespace safe {
    template<typename T, auto Requirement>
    struct var;
}
```

`safe::var` wraps a runtime value with an associated `safe::dsl` requirement describing the set of values it must be contained in. The requirement is used to check the value at runtime or prove at compile-time it is satisfied.

3.1.1. Member constants

3.1.1.1. `requirement`

The `safe::dsl` requirement describing allowed values.

3.1.2. Member functions

3.1.2.1. Constructor

```
constexpr var() requires(requirement >= set<0>);
```

Default constructor, only valid if the requirement allows a value of '0'.

```
template<typename U>
requires(std::is_convertable_v<U, T>)
constexpr var(unsafe_cast_ferry<U> ferry);
```

Unsafe constructor. Used to construct a `safe::var` bypassing all compile-time and runtime checking mechanisms. Leads to undefined behavior if used incorrectly.

```
constexpr var(Var auto const & rhs);
```

Construct a `safe::var` from another instance with potentially different, but compatible requirements. Assignment safety is checked at compile time.

3.1.2.2. operator=

```
constexpr auto operator=(Var auto & rhs) -> var &;
```

Assign value from another instance with potentially different, but compatible requirements. Assignment safety is checked at compile time.

3.1.3. Non-member functions

3.1.3.1. operator<=>

```
[[nodiscard]] constexpr auto operator<=>(Var auto lhs, Var auto rhs);
```

Apply <code>operator<=></code> to <code>lhs</code> and <code>rhs</code> and return the result.

3.1.3.2. operator==

```
[[nodiscard]] constexpr auto operator==(Var auto lhs, Var auto rhs) -> bool;
```

Apply `operator==` to `lhs` and `rhs` and return the result.

3.1.3.3. operator+

```
[[nodiscard]] constexpr auto operator+(Var auto lhs, Var auto rhs);
```

Add the underlying values of `lhs` and `rhs` and return the result.

Value types are promoted to a wider type if the result would otherwise overflow or underflow. No wraparound for signed or unsigned types.

Value types are demoted to a narrower type, down to the natural word size of the underlying architecture, if all possible results will fit.

3.1.3.4. operator-

```
[[nodiscard]] constexpr auto operator-(Var auto lhs, Var auto rhs);
```

Subtract the underlying values of `lhs` and `rhs` and return the result.

Value types are promoted to a wider type if the result would otherwise overflow or underflow. No wraparound for signed or unsigned types.

Value types are demoted to a narrower type, down to the natural word size of the underlying architecture, if all possible results will fit.

```
[[nodiscard]] constexpr auto operator-(Var auto v);
```

Returns `0_i - v`.

3.1.3.5. operator*

```
[[nodiscard]] constexpr auto operator*(Var auto lhs, Var auto rhs);
```

Multiply the underlying values of `lhs` and `rhs` and return the result.

Value types are promoted to a wider type if the result would otherwise overflow or underflow. No wraparound for signed or unsigned types.

3.1.3.6. operator/

```
[[nodiscard]] constexpr auto operator/(Var auto lhs, Var auto rhs);
```

Divide the underlying values of `lhs` and `rhs` and return the result.

Value types are demoted to a narrower type, down to the natural word size of the underlying architecture, if all possible results will fit.

WARNING	A compilation error will result if the `rhs` requirement doesn't exclude '0'.

3.1.3.7. operator%

```
[[nodiscard]] constexpr auto operator%(Var auto lhs, Var auto rhs);
```

Modulo the underlying values of `lhs` and `rhs` and return the result.

Value types are demoted to a narrower type, down to the natural word size of the underlying architecture, if all possible results will fit.

WARNING	A compilation error will result if the `rhs` requirement doesn't exclude '0'.

3.1.3.8. operator<<

```
[[nodiscard]] constexpr auto operator<<(Var auto lhs, Var auto rhs);
```

Shift `lhs` left by `rhs` bit positions and return the result.

Value types are promoted to a wider type if the result would otherwise overflow or underflow. No wraparound for signed or unsigned types.

WARNING	A compilation error will result if the `rhs` requirement doesn't exclude all negative numbers or numbers larger than the bit width of `lhs`.

3.1.3.9. operator>>

```
[[nodiscard]] constexpr auto operator>>(Var auto lhs, Var auto rhs);
```

Shift `lhs` right by `rhs` bit positions and return the result.

Value types are demoted to a narrower type, down to the natural word size of the underlying architecture, if all possible results will fit.

WARNING	A compilation error will result if the `rhs` requirement doesn't exclude all negative numbers or numbers larger than the bit width of `lhs`.

3.1.3.10. operator|

```
[[nodiscard]] constexpr auto operator|(Var auto lhs, Var auto rhs);
```

Bitwise OR the underlying values of `lhs` and `rhs` and return the result.

The resulting value type is the wider type of `lhs` and `rhs`.

3.1.3.11. operator&

```
[[nodiscard]] constexpr auto operator&(Var auto lhs, Var auto rhs);
```

Bitwise AND the underlying values of `lhs` and `rhs` and return the result.

The resulting value type is the narrower type of `lhs` and `rhs`.

3.1.3.12. operator^

```
[[nodiscard]] constexpr auto operator^(Var auto lhs, Var auto rhs);
```

Bitwise XOR the underlying values of `lhs` and `rhs` and return the result.

The resulting value type is the wider type of `lhs` and `rhs`.

3.1.3.13. abs

```
[[nodiscard]] constexpr auto abs(Var auto v);
```

Calculate the absolute value of `v` and return the result.

3.1.3.14. bit_width

```
[[nodiscard]] constexpr auto bit_width(Var auto v);
```

Calculate the bit width of `v` and return the result.

Value types are demoted to a narrower type, down to the natural word size of the underlying architecture, if all possible results will fit.

3.1.3.15. min

```
[[nodiscard]] constexpr auto min(Var auto lhs, Var auto rhs);
```

Calculate the minimum of `lhs` and `rhs` and return the result.

The resulting value type is the narrower type of `lhs` and `rhs`.

3.1.3.16. max

```
[[nodiscard]] constexpr auto max(Var auto lhs, Var auto rhs);
```

Calculate the maximum of `lhs` and `rhs` and return the result.

The resulting value type is the wider type of `lhs` and `rhs`.

3.1.3.17. clamp

```
[[nodiscard]] constexpr auto clamp(
    Var auto value,
    Var auto min_val,
    Var auto max_val
);
```

Clamps `value` from `min_val` to `max_val` and return the result.

The resulting value type is the underlying type of `max_val`.

```
[[nodiscard]] constexpr auto clamp(
    auto unsafe_value,
    Var auto min_val,
    Var auto max_val
);
```

Clamps `value` from `min_val` to `max_val` and return the result.

The resulting value type is the underlying type of `max_val`.

3.2. `safe::dsl`

3.2.1. operator==

```
[[nodiscard]] constexpr auto operator==(auto A, auto B) -> bool;
```

`A = B`

Return true if the set of numbers represented by the requirements `A` and `B` contain exactly the same numbers.

3.2.2. operator!=

```
[[nodiscard]] constexpr auto operator!=(auto A, auto B) -> bool;
```

`A != B`

Return true if the set of numbers represented by the requirements `A` and `B` contain at least one different number.

3.2.3. operator<=

```
[[nodiscard]] constexpr auto operator<=(auto A, auto B) -> bool;
```

`A subseteq B`

Return true if the set of numbers represented by the requirement `rhs` contains all the numbers `lhs` does.

3.2.4. operator>=

```
[[nodiscard]] constexpr auto operator>=(auto A, auto B) -> bool;
```

`A supe B`

Return true if the set of numbers represented by the requirement `lhs` contains all the numbers `rhs` does.

3.2.5. operator||

```
[[nodiscard]] constexpr auto operator||(auto A, auto B) -> bool;
```

`A uu B`

3.2.6. operator&&

```
[[nodiscard]] constexpr auto operator&&(auto A, auto B) -> bool;
```

`A nn B`

3.2.7. operator+

```
[[nodiscard]] constexpr auto operator+(auto A, auto B);
```

`{a + b \| a in A, b in B}`

3.2.8. operator-

```
[[nodiscard]] constexpr auto operator-(auto A, auto B);
```

`{a - b \| a in A, b in B}`

3.2.9. operator*

```
[[nodiscard]] constexpr auto operator*(auto A, auto B);
```

{a * b \| a in A, b in B}

3.2.10. operator/

```
[[nodiscard]] constexpr auto operator/(auto A, auto B);
```

{a / b \| a in A, b in B}

3.2.11. operator%

```
[[nodiscard]] constexpr auto operator/(auto A, auto B);
```

{a % b \| a in A, b in B}

3.2.12. abs

```
[[nodiscard]] constexpr auto abs(auto A);
```

{\|a\| \| a in A}

3.2.13. min

```
[[nodiscard]] constexpr auto min(auto A, auto B);
```

{min(a, b) \| a in A, b in B}

3.2.14. max

```
[[nodiscard]] constexpr auto max(auto A, auto B);
```

{max(a, b) \| a in A, b in B}

3.2.15. operator&

```
[[nodiscard]] constexpr auto operator&(auto A, auto B);
```

{a\ &\ b \| a in A, b in B}

3.2.16. operator|

```
[[nodiscard]] constexpr auto operator|(auto A, auto B);
```

```
{a \| b \ \|\ a in A, b in B}
```

3.2.17. operator^

```
[[nodiscard]] constexpr auto operator^(auto A, auto B);
```

```
{a\ o+\ b \| a in A, b in B}
```

3.2.18. operator~

```
[[nodiscard]] constexpr auto operator~(auto A);
```

```
{~a \| a in A}
```

3.2.19. operator<<

```
[[nodiscard]] constexpr auto operator<<(auto A, auto B);
```

```
{a " << " b \| a in A, b in B}
```

3.2.20. operator>>

```
[[nodiscard]] constexpr auto operator>>(auto A, auto B);
```

```
{a " >> " b \| a in A, b in B}
```

3.3. `safe::array`

3.4. `safe::integer`

3.5. Algorithms

4. Requirements DSL Theory of Operation

4.1. Subset is Left Distributive Over Union of Disjoint Intervals

For intervals A, B, and C, where B and C are disjoint, the following holds true:

`A subseteq (B uu C) -= (A subseteq B) vee (A subseteq C)`

1	Given `A subseteq (B uu C)`	
2	`x in A -> x in (B uu C)`	Definition of subset
3	`x in A -> (x in B vee x in C)`	Definition of union
4	`(x in A -> x in B) vee (x in A -> x in C)`	Implication is left distributive over disjunction
5	`(A subseteq B) vee (A subseteq C)`	Definition of subset
6	`therefore A subseteq (B uu C) -= (A subseteq B) vee (A subseteq C)`	

4.2. Subset is Right Anti-Distributive over Union of Disjoint Intervals

For intervals A, B, and C, where B and C are disjoint, the following holds true:

`(A uu B) subseteq C -= (A subseteq C) wedge (B subseteq C)`

1	Given `(A uu B) subseteq C`	
2	`x in (A uu B) -> x in C`	Definition of subset
3	`(x in A vee x in B) -> x in C`	Definition of union
4	`(x in A -> x in C) wedge (x in B -> x in C)`	Implication is right anti-distributive over disjunction
5	`(A subseteq C) wedge (B subseteq C)`	Definition of subset
6	`therefore (A uu B) subseteq C -= (A subseteq C) wedge (B subseteq C)`	

4.3. Subsets of Unions of Disjoint Intervals

For disjoint intervals A and B, and disjoint intervals C and D, the following holds true:

`(A uu B) subseteq (C uu D) -= {(A subseteq C) vee (A subseteq D)} wedge {(B subseteq C) vee (B subseteq D)}`

1	Given `(A uu B) subseteq (C uu D)`	
2	`{A subseteq (C uu D)} wedge {B subseteq (C uu D)}`	Subset is Right Anti-Distributive over Union of Disjoint Intervals
3	`{(A subseteq C) vee (A subseteq D)} wedge {(B subseteq C) vee (B subseteq D)}`	Subset is Left Distributive Over Union of Disjoint Intervals
4	`therefore (A uu B) subseteq (C uu D) -= {(A subseteq C) vee (A subseteq D)} wedge {(B subseteq C) vee (B subseteq D)}`	

4.4. Cartesian Product of Unions

`(A uu B) xx (C uu D) -= (A xx C) uu (B xx D) uu (A xx D) uu (B xx C)`

https://proofwiki.org/wiki/Cartesian_Product_of_Unions

4.5. Cartesian Product is Distributive Over Union

`A xx (B uu C) -= (A xx B) uu (A xx C)`

`(B uu C) xx A -= (B xx A) uu (C xx A)`

https://proofwiki.org/wiki/Cartesian_Product_Distributes_over_Union