SCALE Rust Documentation

B. Coenen A. Mertens O. Scherer N.P. Smart

August 27, 2021

Contents

1 Changes

This is v1.3 of the Rust-based implementation system for SCALE.

1.1 Current Known Bugs

1. Operator arithmetic operations for `ClearInteger<K>` and `SecretInteger<K>`.
2. Functions with multiple return values can cause a problem for the compiler, especially when they are not-inlined. Thus avoid these. If you get weird behaviour this might be the reason.
3. When compiling some loops the Rust-to-Wasm compiler sometimes creates registers which are 'assumed' to be zero-assigned on first use. No idea why this happens, but you can see it by compiling SCALE with the flag `-DDEBUG`. This seems to cause no effect for normal programs, but will cause problems (possibly) for programs called via the `restart` mechanism. To avoid this problem we issue a `clear_register` command at the start of each program, and a Rust `restart` command deletes the memory allocated for testing and the wasm stack before executing the SCALE `restart` opcode.

1.2 Changes in version 1.3 from 1.2

1. A little more of the emulation, in the testing mode, is correct. This aids a bit with debugging.
2. A bug related to function calling has been fixed.
3. Optimized `SecretBit` operations.
4. $\sqrt{x}$ and $\log(x)$ operations have been implemented for `ClearFloat` and `SecretFloat` data types. Note, this is a definite improvement on Mamba, where they were not implemented at all.
5. A simple version of ORAM style operation is now implemented. This enables reading/writing to `Array`s and `Slice`s given a $\mathbf{SecretModp}$ index.
6. We can now support `Array`s and `Slice`s of `ClearFloat` and `SecretFloat` values. By extension by copying how the allocators are done for these types, one can also support arrays and slices of arbitrary types which consist of multiple elements of the same base-type. Eventually this needs to be extended to types which consist of elements of different base types.
7. You no longer need to define channels and players using the keywords `Channel` and `Player`. This means you channels and players are normal `i64` values, and hence can be programmatically defined, i.e. via loop values etc.

1.3 Changes in version 1.2 from 1.1

1. Decision has been made to not port the vectorized instructions over to Rust. The reason is that they require changes to the assembler, the compiler *and* require the programmer to jump through hoops to use them. Instead we are implementing optimization pipelines and changes to the runtime to allow the same 'effects', at time critical components, to be applied to programs without the need for the user to explicitly work this out and program this themselves. The first change is a merging of `TRIPLE` and `SQUARE` instructions, which is oblivious to the user, but gives for some programs a ten percent performance improvement.
2. The second change is a 'vectorized' private input and output routine for Arrays and Slices, which under-the-hood works via the SCALE memory (as opposed to the usual SCALE method of vectorizations by having contiguous register names).
3. The third change is some more advanced operations on Arrays and Slices, which have been propogated into our Rust standard library.

4. The overall effect is that programs in the Rust pipeline now are about as efficient as the equivalent programs in the Mamba pipeline.

5. Added data types `ClearFloat<V,P>` and `SecretFloat<V,P>` to replicate the cfloat and sfloat in Mamba. The maths library for these datatypes is not yet fully implemented.

6. Almost all data types now have a means for generating random values within them.

7. Some operations on integers and bit operations have had their API changed a little.

8. Modified the way `restart()` is compiled to avoid a bug when using the restart functionality and rust programs, for a similar reason we issue a `clear_register` operation now at the start of each program. See the 'Known Bugs' section above. (NPS:) *There is still a bug here. Will fix soon*

1.4 Changes in version 1.1 from 1.0

1. Some functions which should have been **unsafe** are now marked as **unsafe**.

2. Added documentation of some functions related to bit processing.

3. Added in comparison member functions for the `ClearInteger<K>` type.

4. Fixed some arithmetic bugs in `ClearInteger<K>` and `SecretInteger<K>` types.

5. Round complexity between Rust pipeline and the old Mamba pipeline is the same now for some test benchmarking functions.

6. The `ClearIEEE` class can now process arithmetic operations as well as having a full math library, which is relatively fast. Thus the `ClearIEEE` type should be preferred for all *clear* operations on floating point values.

7. A full math library for `SecretIEEE` is implemented.

8. The crate for `ClearIEEE` and `SecretIEEE` is now called simply `scale_std::ieee`

9. Added fixed point operations with new data types `ClearFixed<K,F>` and `SecretFixed<K,F>`. Note, the `SecretFixed<K,F>` is generally much faster than the `SecretIEEE` type.

10. Conversion between `ClearFixed<K,F>` and `ClearIEEE` is possible by means of the respective `from` functions.

11. Arrays/Slices can now be created for the datatypes `ClearIEEE`, `SecretIEEE`, `ClearFixed<K,F>` and `SecretFixed<K,F>`.

12. Memory management is now dynamic and done via the SCALE runtime, thus the need to define memory size at the start of a SCALE Rust program has gone.

13. `Array` and `Slice` are now more robust and closer to what the standard Rust `Vec` type does. Note use of `get` on an `Array` and `Slice` costs at runtime so use `get_unchecked` where possible.

2 Introduction

2.1 Installation

You **may** need to be on the Rust nightly build and you probably also need to add `wasm` support by typing

```
rustup target add wasm32-unknown-unknown
```

2.2 General Rust Programs

Programs live in folders (just like in Mamba), however the location of the Rust program folders is now in

```
RustPrograms/examples/<program name>/
```

The main Rust program in this folder should be called `main.rs`. To compile the program in the above folder you execute

```
./compile-rust.sh <program name>
```

This will execute the compiler and place the associated `.bc` and `.sch` files within the above folder. To run the program you then have to execute

```
./Player.x 0 RustPrograms/examples/<program name>/
```

There are various options you can pass to the compiler, these include

- `-A`. This places the associated Scale assembly files `.asm` in the program folder. These are the assembly files *before* they have been passed through the `scasm` optimization steps.

Any other options (like `-O1, -O2, -O3`) get passed to scasm.

Your Rust programs should start with the following preamble:

```
#![no_std]
#![no_main]
#![feature(const_evaluatable_checked)]

#[scale::main(KAPPA = 40)]
#[inline(always)]
fn main() {
    // your code goes here
}
```

The `main!` part defines the `KAPPA` parameter, which is the global statistical security parameter which your program will use. Unlike in Mamba we define a single such parameter which is used for all operations; e.g. the `SecretInteger` and `SecretFixed` types (and for the `sfloat` when it ends up being defined).

2.3 Testing

The compiler can compile into two different targets, the real SCALE target, which is done as above, and a Test target, which is used to test the system. In the test system the Rust program is simply run on a single machine, and all secret operations are performed in the clear. The other difference between the two modes is in the operation of the `test` operations which are given later. In the SCALE target they create a write to memory, in the Test target they read from the memory output by the SCALE execution and then compare this value to the value which is being tested.

Note: The test instructions need to occur on a well defined, unique, line. Thus you cannot place a test function call within a loop. All test operations should be executed within the main function. The line number needs to be less than 1000; if you execute a test instruction on a line number greater than 1000 then undefined behaviour can occur.

To compile and run the test target you invoke

```
./Scripts/test_wasm.sh <program name>
```

which will take all `*.rs` files in `RustPrograms/examples/<program name>`, convert them to `wasm`, and then to scale assembly and then run them in the SCALE test environment. After that it will run the same programs in the clear and compare their approprate `test` calls.

To compile and run all the test programs on the current configuration defined by `Setup.x` use

```
./Scripts/test_wasm.sh
```

To compile and run all the test programs on all the test configurations you invoke

```
./run_tests.sh test_wasm
```

2.4 Things Working/Not Working

The advantage of using the Rust over Mamba driving language is that we now have a means to start defining more complex operations in a clearer manner. One should think of Mamba not as a language, but as a high-level assembler.

2.4.1 Not Working

Currently you cannot access from Rust the following features of Scale

- More than one online thread.
- Some native SCALE printing operations.
- `sfloat`, `cfloat`.

We are currently working on fixing the above.

The vectorized (i.e. SIMD) instructions in SCALE will not be supported by the Rust pipeline. To access them the programmer would need to use special constructs etc, which make using them difficult. The gain is marginal, over program complexity, and we can achieve the same effect by other changes in the runtime which can apply to a wider class of programs.

2.4.2 Working

In comparison to Mamba the following work or are improvements in our Rust compilation pipeline.

- You cannot directly access comparison etc operations on `ClearModp`/`SecretModp` types, after all what does comparison of two integers modulo p mean mathematically? Instead these operations are available via the `ClearInteger<K>`/`SecretInteger<K>` types. Thus you need to coerce your value into one of these types first (which costs nothing), and then apply the desired operation. This avoids programmer errors, as the programmer needs to know how big an integer is before it can be compared to another.
- Recursive function calls.
- Automatic assignment of mutable local variables across conditional basic blocks; avoiding the need to manually work out memory allocations.
- No distinction between the old 'python'-for loops (unrolled) vs 'Mamba'-for loops (rolled). The distinction is determined by the compiler as an optimization.
- Conveniently allocating memory instead of having to keep track of it manually in the code.
- Dynamic memory allocation and deletion for arrays etc is done automatically.
- Full math library for the `ClearIEEE` and `SecretIEEE` types (none of which are available in Mamba).

- In working through the code we found lots of assumptions which Mamba makes which are not strictly true. For example when doing integer comparison of objects modulo p it (sometimes) assumes that the underlying integer is less than 64-bits in size. Whilst for a lot of code this might be perfectly correct, for some use cases it is not (for example fixed point arithmetic of high precision for large numbers). Thus when making the libraries for arithmetic we have assumed all possible use cases are possible. Thus code may be less efficient than Mamba, but it will be hopefully more correct.

2.5 Under The Hood: How the magic works...

Wasm is a stack based assembly language. There are no registers. Instead, all instruction arguments are pushed and popped on a stack. As an example, take $42 + 3$ in `wasm`. First 42 is pushed to the stack, then 3 is pushed to the stack and finally add pops two values off the stack, adds them and pushes the result back onto the stack.

```
i32.const 42
i32.const 3
i32.add
```

Scale is a register based assembly language. While it does have a stack, meaning we could emulate `wasm` 1:1 by generating

```
ldint r0 42
pushint r0

ldint r0 3
pushint r0

popint r1
popint r2
addint r3 r1 r2
pushint r3
```

But that would be very inefficient.

Basic Transpilation What we do instead is to essentially execute the `wasm`, just like mamba did with its source. So when we see

```
i32.const 42
```

we push an `Operand::Value(Const::Int(42))` to the wasca-stack. The wasca-stack only exists during transpilation and is entirely unrelated to the `push*` and `pop*` operations from scale.

When we see

```
i32.add
```

We pop two values off the wasca-stack. If the values are `Operand::Value` and not `Operand::Register`, we begin by generating

```
ldint some_new_register popped_operand_value
```

This conversion is automatically done by the `val_to_reg` function. The $some_new_register$ is 'allocated' by increasing a global counter in wasca. This way we always get a new register and guarantee that all registers are used in a SSA manner.

Now that we have two `Operand::Register` (either directly from popping or from the conversion) we allocate a result register and generate the add instruction.

```
addint result_reg reg1 reg2
```

Finally, we push an `Operand::Register` for the result register onto the wasca-stack. The next operation can then pop the value off the stack in order to obtain its arguments.

2.5.1 Local variables

Sometimes a stack based approach makes certain code very complex or nearly impossible to write. For this reason `wasm` has function-local variables. Every function declares at the start which local variables it has and of which type they are. Unfortunately `wasm` only has `i32 i64 f32 f64 i128` available to us. Technically there is also an externref meta-type that allows code to create its own types, but LLVM (and thus Rust) does not support this yet. This means that while we have 5 `wasm` types that could map to the five SCALE types (where here we use their Rust names) `i64 SecretI64 ClearModp SecretModp SecretBit`. The `wasm` types have no relation to the types used from Rust, beyond that `i64` in Rust is also `i64` in `wasm`. Unfortunately `u64` in Rust is also `i64` in `wasm`, and the only way to notice this in `wasm` is that there are some unsigned operations on `i64`. We can ignore all this for most of the time.

The interesting part is that `SecretModp`, `SecretI64`, and `ClearModp` are all encoded as `f64`, `f32` and `i128`, without ever using them as such. The reason this works out at all is that we do not translate operations on these types to `wasm` operations, but instead translate them to extern function calls which we will lower to the appropriate instructions directly. `SecretBit` is encoded as `SecretI64` by being converted to and from `SecretI64` at all use sites. This is not very efficient, but we will get efficiency back once LLVM and Rust permit us to declare externref types.

2.5.2 Extern function calls

Operations on anything but `i64` as well as any SCALE instructions that just does not have an equivalent in `wasm` are represented as extern function calls. As an example, take the adds instruction. Its arguments as per the documentation are `(dest: sw, left: s, right: s)`. From this, the transpiler automatically figures out that there is one return (output) value and two arguments. In Rust, this generates a

```
extern "C" {
    fn __adds(left: Secret, right: Secret) -> Secret;
}
```

Any call to `__adds` will cause the `wasm` code to leave two values on the `wasm` stack and expect that after the call the return value will be on the `wasm` stack. So, similarly to the builtin addition for `i64`, we will pop two values off the stack, make sure they are registers by optionally generating `lds` instructions and then push the return register onto the stack.

Some SCALE instructions have multiple output/return values. Examples are the triple and square instructions. Unfortunately Rust/LLVM does not really support this yet, so we had to work around it a bit. For these functions we generate a `push_multi_arg_triple` function which has no return values at all. This function pushes values onto the transpiler stack, even though the `wasm` stack would not contain any values. **Remember:** The transpiler stack has nothing to do with the SCALE stack at all, it is just equivalent to the `wasm` stack most of the time, unless we do hacks as described in this section. Now, after these values are on the stack, we use transpiler stack manipulation functions (extern functions by themselves) to get the values back. In essence, the `__triple` 'extern' function is thus not an extern function but looks like

```
#[inline(always)]
pub fn __triple() -> (Secret, Secret, Secret) {
    push_multi_arg_triple();
    let a = pop_multiarg();
    let b = pop_multiarg();
    let c = pop_multiarg();
    (a, b, c)
}
```

Function calls: Most function calls in Rust do not actually end up with a function call in `wasm`. This is because usually the optimizer inlines function calls aggressively. In some cases we do want function calls instead of aggressive inlining. In the future there will be a way to force the compiler to generate a function call at specific sites.

2.5.3 Memory Management

Memory is allocated by use of the `New` and `Delete` operations in the SCALE runtime. This is mainly hidden from the user, and the user should probably not use these (just as a C++ programmer should probably avoid `new` and `delete` and stick to the STL).

For single value allocations, we have the `scale_std::heap::Box` that works similarly to Array, but has no indices on its operations. We should analyze whether we need any other memory datastructures and how much users should be able to nest them (array of slices of boxes or similar).

Right now we use regular Rust globals for the memory allocator. This has the disadvantage of requiring memory itself (in the compiler-used memory region). This makes various operations very verbose (10-20 instructions for a single memor allcation invocation). If we add special functions that handle the memory allocations for us in wasca, we can optimize all this to just use a single register per memory bank.

3 Rust Dummies Guide

Using Rust has some advantages in that we have strong typing and the programming language has been designed to be safe in terms of stopping programmers making errors. However, these advantages come at a disadvantage (especially if you are used to Python or C++). So in this section we give an informal overview of how these differences affect programming in our Rust system, and in particular with relevance to SCALE.

3.1 Strong Typing

We take advantage of strong typing to ensure that the types correspond to mathematically what they represent. For efficiency we give access to the basic SCALE internal types **i64**, **ClearModp**, **SecretModp**, **SecretI64** and **SecretBit**. But in MAMBA you could do something like

```
program.bit_length = 16
program.security = 40

a = sint(4)
b = sint(5)
c = a<b
```

But this is mathematically meaningless, what is meant here is that you are going to 'think' of `sint` values as integers of bit size sixteen and then do the comparison assuming that the numbers are indeed of size bounded by 2^{16}. After all a comparison of values in the finite field $\mathbb{F}_p$ really makes no mathematical sense.

In our Rust system we want to avoid such implicit assumptions that a reader of your code needs to make. Thus we provide types which help capture what you really want to do. So the above MAMBA code would become

```
let a: SecretInteger<16> = SecretInteger::from(4);
let b: SecretInteger<16> = SecretInteger::from(5);
let c = a.lt(b);
```

The type of the value `c` is a **SecretModp** value. We use the member function notation `a.lt(b)` instead of `a<b` to force the programmer to realise that you cannot use the output of the comparison in an `if`-statement.

Another aspect of this strong typing is the above use of the `from` command. This is used to convert one type to another, which needs to be done explicitly in almost all case.

3.2 Mutable vs Non-Mutable

Common problems in programs are that people accidentally re-assign a variable and then want the old value again. This is because in languages like C++ or python all variables are mutable by default. In C++ it is usually considered good practice to define all inputs to a function to be `const` if they are not going to be returned as changed for this reason. Rust goes one step further and assumes all variables are non-mutable by default. Thus you cannot do

```
let a = 3;
if some_condition {
    a = a + 1;
}
```

To enable this you explicitly have to signal that the variable is going to be changed by writing

```
let mut a = 3;
if some_condition {
    a = a + 1;
}
```

3.3 Container Types

Container types are really important in programming, yet in Rust they can be a bit confusing mainly due to the mutability issue above, and the need to maintain safety of the language. The `Array` and `Slice` types we define are very similar to the standard `Vec` type in Rust, and hopefully eventually they will be the same. The difference between an `Array` and a `Slice` is that the size of an `Array` is known at compile time, whereas a `Slice` size may not. This distinction enables us to do some optimizations.

Suppose you have an array of ten **ClearModp** values

```
let mut a: Array<ClearModp, 10> = Array::uninitialized();
```

You may want to assign values to the array elements, or use them later. There are *four* ways of getting an array element:

1. `A.get(i)` returns a reference to the array and performs an out of bounds check.
2. `A.get_unchecked(i)` returns a reference to the array and does not perform an out of bounds check.
3. `A.get_mut(i)` returns a mutable reference to the array and performs an out of bounds check.
4. `A.get_mut_unchecked(i)` returns a mutable reference to the array and does not perform an out of bounds check.

When accessing the reference it comes wrapped in an `Option` when using the checked values, thus you need to `unwrap` this option before using the item. The unwrapping itself produces a guarded object, to remove the `Guard` you need to de-reference it.

Thus when you use a `get`, but not a `get_unchecked`, you need to unwrap the result, then in both cases you should de-reference, i.e. you do

```
println!("␣a[2]␣=␣",*a.get(2).unwrap());
println!("␣a[2]␣=␣",*a.get_unchecked(2));
```

However, for printing we have added some code to make the following more syntactically nicer code work,

```
println!("␣a[2]␣=␣",a.get(2).unwrap());
println!("␣a[2]␣=␣",a.get_unchecked(2));
```

If you want to access an element *and not change it* you should use one of the non-mutable `get` operations, if you want to change an element you should access one of the mutable, i.e. `get_mut`, operations. This is particularly relevant when the internal object is another `Array` or `Slice`.

```
let mut a: Slice<Array<i64, 2>> = Slice::uninitialized(5);
for i in 0..5 {
    for j in 0..2 {
        a.get_mut(i).unwrap().set(j, &((i * 2 + j) as i64));
    }
}
```

To modify elements in a simple `Array` or `Slice` use.

```
let mut a: Array<ClearModp, 10> = Array::uninitialized();
a.set(2, &ClearModp::from(1));
a.set(3, &ClearModp::from(4));
```

Now suppose you have a `Slice` of `Arrays`

```
let mut S: Slice<Array<ClearModp, 2>> = Slice::uninitialized(5);
```

and after some processing you would like to take the fourth element of the `Slice`.

```
let mut A = *S.get_mut(4).unwrap();
```

The value A is now an Array of length two. What you really want is for A to be a Copy of S. But not all types in Rust enable copying, whilst our basic types do the Array and Slice types do not. Thus in C++ terms in the above code the A value is really just a 'pointer' to the fourth Array in the Slice S. Thus the effect would be that if you changed elements in A then you also change the elements in S. In addition if S goes out of scope and gets deleted then so will A.

To avoid this problem you need to clone the output of the get as in

```
let mut A = S.get(4).unwrap().clone();
```

But you only need to do this as the Array type is not copyable. If you had the following

```
let mut A : Array<ClearModp, 10> = Array::uninitialized();
a.set(3, ClearModp::from(4));
let mut a = *A.get(3).unwrap();
a = a + 3;
```

then a really is a copy of the entry in A. So at the end we have $a = 7$ and $A[3] = 4$.

The unchecked versions of the get operations should only be used if you know what you are doing. However, the checked versions do have a performance cost; they require a run-time branch which may impact the optimizers ability to reduce the total number of rounds of communication.

For more details on Options see

- https://doc.rust-lang.org/std/option/

4 SCALE Dummies Guide

There are a lot of types defined here, and to exploit the most from the system you need to (kind of) understand how the MPC engine works under-the-hood. As most people will not be able to do that we here provide a quick dummy guide to get the most performance out of the system. In this section we are mainly talking about your programs, and not system tuning for an application. The latter topic is far more of a black-art alas.

4.1 Integer Types

There two *basic* clear integer types **i64** and `ClearModp`. The former represent 64-bit integers and the latter integers modulo p, for the prime you have chosen. These work at roughly machine speed, bar any overhead due to the VM. There is an *advanced* clear integer types `Integer<K>` which holds an integer in $\mathbb{Z}$ as a `ClearModp` but with the 'soft' gaurantee that the integer lies in $\mathbb{Z}_{\langle k \rangle}$. On the secret side these are duplicated by the `SecretI64`, `SecretModp` and `SecretInteger<K>` types.

Note the secret operations are *always* more expensive than the clear operations, and that `SecretI64` operations are generally more expensive than `SecretModp` operations. The `SecretInteger<K>` type is more costly than a `SecretModp` type, as it has to worry about size issues. Conversion between `SecretModp` and `SecretI64` types are expensive, thus best avoided if possible. Conversion betwen `SecretModp` and `SecretInteger<K>` types if for free, as it is assumed the programmer knows the conversion will be 'valid'.

But there seems to be more arithmetic operations for `SecretI64` than for `SecretModp`. For example, you cannot compare two `SecretModp` values (after all integers modulo p have no notion of 'size', so mathematically this makes no sense).

However, in MPC algorithms we often use `SecretModp` values to hold integers (i.e. elements of $\mathbb{Z}$) and compute on them whilst keeping a mental note of how big they are. This means we can perform operations by keeping in the `SecretModp` domain, without needing to convert to `SecretI64` values.

```
let a = SecretModp::from(10);
let b = SecretModp::from(20);

let c = a*b;
let d = a+b;

// Suppose we now want to "compare" c and d.
// We know the max size is 9 bits (c is 200,
// which is at least 9 bits in signed representation)
let ci : SecretInteger<9> = SecretInteger::from(c);
let di : SecretInteger<9> = SecretInteger::from(d);
let compare = ci.lt(di);
```

The above is *much* faster than mapping `c` and `d` over to the `SecretI64` domain and doing the comparison there. Obviously the above code is a bit fake, as you know what the values are, but you can see the idea.

4.2 Floating Point Types

In general `ClearIEEE` should be your preferred clear floating point type, as it provides almost machine level performance. The preferred secret floating point type is `SecretFixed<K,F>` as it is much faster. However, it only provides fixed point arithmetic. If true floating point is required then `SecretFloat<V,P>` is better, but it can result in programs which take ages to compile. For faster compilation, but slower programs, use `SecretIEEE`

4.3 Bits

There is a `SecretBit` type. Due to some current limitations of the intermediate `wasm` representation there is a lot of conversions between `SecretBit` types and `SecretModp` types under the hood. This results in slower programs

than are really necessary. Once `wasm` has been extended to cope with more than four basic register types this restriction will be removed.

5 Basic Rust Types

The main thing to keep in mind is that **currently** the only Rust type supported is the **i64** type. However, if you **use** another type it might cause neither an error nor a crash of the compiler (at the moment, we want to fix this); so you might get undefined behaviour.

While it works to just use integral constants almost everywhere, it can be more efficient to use the `ConstI32` and `ConstU32` values instead. This is because for example `ClearModp::from(42)` will end up creating two assembly instructions, one to load the 42 into a regint register, the second one for a conversion from regint to cint. If `ClearModp::from(ConstI32::<42>)` is used instead, you get just one assembly instruction to directly load a constant into a cint register.

Similarly there are operations which are only supported on constants. One example is dividing a `SecretModp` by an integer, this only works when using `ConstI32`, as MPC has a special (and thus very efficient) instruction for this kind of division.

All of the basic SCALE types in Rust support the `Copy` and `Clone` traits.

5.1 i64

The Rust type **i64** corresponds to the regint type in the underlying SCALE virtual machine. You create a value of type **i64** (everything in this document also applies to `u64`) by appending `_i64` to a number, like in `42_i64`. Mostly you do not need this postfix, but sometimes the compiler cannot figure out that you meant to use an **i64** and may thus give you type mismatch errors because it used a fallback to `i32`.

5.1.1 Conversion of Data

`i64::from(x)`

You can convert from a **ClearModp** value to an **i64**.

```
let ca = ClearModp::from(10);
let a=i64::from(ca);
```

5.1.2 IO

The IO class functions for **i64** datatypes can be access via the following commands:

`i64::input(x)`

Loads an **i64** from the channel x.

```
let a=i64::input(10);
```

`a.output(x)`

Writes an **i64** to channel x.

```
let v: i64 = 1;
v.output(10);
```

5.1.3 Comparisons

The following 'operators' can be applied between two **i64** values or a **i64** value and a **SecretI64** value. The output being an **i64** value if the two arguments are **i64**, and a **SecretBit** value otherwise. As the result of the operator (when a **SecretBit**) cannot be used in a conditional branch, we use the member function notation for such 'operators', as opposed to the operator notation. Thus syntactically the programmer is less likely to make a mistake.

a.gt(x), a.ge(x), a.lt(x), a.le(x), a.gt(x), a.eq(x), a.ne(x)

5.1.4 Other Functions

x.rand()

Produces a pseudo-random number in the range $[0, \ldots, x-1]$. Note, every player gets the same value; this is used for randomized algorithms where the algorithm needs to make a random choice.

```
let r = x.rand();
```

i64::randomize()

Produces a random number in the range $[0, \ldots, 2^{64}-1]$. This random number is the 'same' for all players.

```
let a = i64::randomize();
```

5.1.5 Memory Access

a.store_in_mem(x)

To store data into memory location 50 say you need to execute, recalling that memory is accessed by 32 bit values.

```
unsafe{ a.store_in_mem(50_u32); }
```

To store to a variable location you need to ensure the x really is an **i64** by doing:

```
let x: i64 = 50;
unsafe{ a.store_in_mem(x); }
```

i64::load_from_mem(x)

To load data from memory you do

```
let a=i64::load_from_mem(50_u32);
let x: i64 = 50;
let b=i64::load_from_mem(x);
```

5.1.6 Testing Data

black_box(x)

If you just use `let a = 1` then the Rust compiler can optimize the variable away. If you really want to treat a as a `regint` value in the SCALE runtime then use

```
let a = black_box(1);
```

x.test()

In the SCALE target this writes the regint value x into memory at the address equivalent to the line number in which `test` was invoked.

In the Test target this takes a value stored in the regint memory saved on the last SCALE invocation, and compares it to x. If the two values are the same it prints the value, and the line number in the rust file where this was executed. Otherwise it aborts.

`x.test_mem(loc)`

This compares the value held in variable x to the memory location at position loc, a simple example being

```
let c: i64 = 8;
unsafe{ c.store_in_mem(3_u32); }
let d: i64 = 8;
d.test_mem(3_u32);
```

On the SCALE target this is basically a no-op, on the Test target it does this comparison.

`x.test_value(y)`

Test whether the value held in x is the same as the value held in y. This is the same as `x.test()` except the value is compared to y and not the value emulated in the test environment.

5.2 SecretI64

This is the same as the sregint in the underlying SCALE virtual machine.

5.2.1 Conversion of Data

`SecretI64::from(a)`

This takes an integer value a and loads it into a value of type **SecretI64**. You can also pass in an argument of a **SecretModp** or a **SecretBit**.

```
let sa = SecretI64::from(ConstI32::<8>);

let sb = SecretBit::from(false);
let ssb = SecretI64::from(sb);

// See the main SCALE manual for the semantics of this conversion
let si = SecretModp::from(ConstI32::<10>);
let ssi = SecretI64::from(si);
```

5.2.2 Arithmetic

`SecretI64::mult2(a,b)`

One can access a multiplication operation which produces a double word output as follows:

```
let s64a = SecretI64::from(0x4000000000000000_i64+28731371);
let s64b = SecretI64::from(0x4000000000000000_i64+985724131);
let (high,low)=SecretI64::mult2(s64a, s64b);
```

5.2.3 Bit Twiddling

`a.set_bit(b,n)`

This sets the n-th bit of a equal to b.

```
let bit = SecretBit::from(true);
let sa = SecretI64::from(ConstI32::<0>);
let sb=sa.set_bit(bit, ConstU32::<10>);
```

`a.get_bit(n)`

This returns the n-th bit of a as a **SecretBit**.

```
let sa = SecretI64::from(123121);
let sb=sb.get_bit(ConstU32::<10>);
```

5.2.4 Comparisons

The following 'operators' can be applied between two **SecretI64** values or a **SecretI64** value and a **i64** value. The output is a **SecretBit** value. As the result of the operator cannot be used in a conditional branch, we use the member function notation for such 'operators', as opposed to the operator notation. Thus syntactically the programmer is less likely to make a mistake.

a.gt(x), a.ge(x), a.lt(x), a.le(x), a.gt(x), a.eq(x), a.ne(x)

The following give variants which compare just to zero.

a.gtz(), a.gez(), a.ltz(), a.lez(), a.gtz(), a.eqz(), a.nez(x)

5.2.5 Other Functions

SecretI64::randomize()

Produces a (secret) random number in the range $[0, \ldots, 2^{64} - 1]$. This random number is the 'same' for all players.

```
let a = SecretI64::randomize();
```

5.2.6 Memory Access

a.store_in_mem(x)

To store data into memory location 50 say you need to execute, recalling that memory is accessed by 32 bit values.

```
unsafe{ sa.store_in_mem(ConstU32::<50>); }
```

To store to a variable location you use

```
let x: i64 = 50;
unsafe{ sa.store_in_mem(x); }
```

SecretI64::load_from_mem(x)

To load data from memory you do

```
let sa=SecretI64::load_from_mem(ConstU32::<50>);
let x: i64 = 50;
let sb=SecretI64::load_from_mem(x);
```

5.2.7 Testing Data

x.test()

In the SCALE target this takes the **SecretI64** value x, applies a reveal operation to it, and then writes the resulting regint value into memory.

In the Test target this takes a value stored in the regint memory saved on the last SCALE invocation, and compares it to x. If the two values are the same it prints the value, and the line number in the rust file where this was executed. Otherwise it aborts.

x.test_value(y)

Test whether the value held in x is the same as the value held in y. This is the same as `x.test()` except the value is compared to y and not the value emulated in the test environment.

5.3 ClearModp

This is the same as the cint in the underlying SCALE virtual machine.

5.3.1 Conversion of Data

`ClearModp::from(a)`

This takes a constant a, a runtime integer $i64$ or a **i64** and loads it into a value of type **ClearModp**.

```
let ca = CleadModp::from(ConstI32::<8>);
// Works, but creates multiple assembly instructions.
let cx = CleadModp::from(42);
```

5.3.2 IO

The IO class functions for **ClearModp** datatypes can be access via the following commands:

`ClearModp::input(x)`

Loads an **ClearModp** from the channel x.

```
let a=ClearModp::input(10);
```

`a.output(x)`

Writes an **ClearModp** to channel x.

```
let v = ClearModp::from(ConstI32::<1>);
v.output(10);
```

5.3.3 Other Operations

`x.lengendre()`

Computes the legendre symbol of x modulo the prime underlying the **ClearModp** type

```
let l = x.legendre();
```

`x.digest()`

Computes a pseudo-random digest of the value x.

```
let d = x.digest();
```

`ClearModp::randomize()`

Produces a random number in the range $[0, \ldots, p-1]$, where p is the prime underlying the **ClearModp** type. This random number is the 'same' for all players.

```
let a = ClearModp::randomize();
```

5.3.4 Memory Access

`a.store_in_mem(x)`

To store data into memory location 50 say you need to execute, recalling that memory is accessed by 32 bit values.

```
unsafe{ ca.store_in_mem(ConstU32::<50>); }
```

To store to a variable location you use

```
let x: i64 = 50;
unsafe{ ca.store_in_mem(x); }
```

`ClearModp::load_from_mem(x)`

To load data from memory you do

```
let ca=ClearModp::load_from_mem(ConstU32::<50>);
let x: i64 = 50;
cb=ClearModp::load_from_mem(x);
```

5.3.5 Testing Data

`x.test()`

In the SCALE target this takes the **ClearModp** value x, applies a reveal operation to it, and then writes the resulting **ClearModp** value into memory.

In the Test target this takes a value stored in the **ClearModp** memory saved on the last SCALE invocation, and compares it to x. If the two values are the same it prints the value, and the line number in the rust file where this was executed. Otherwise it aborts.

`x.test_mem(loc)`

This compares the value held in variable x to the memory location at position loc, a simple example being

```
let c = ClearModp::from(ConstI32::<8>);
unsafe{ c.store_in_mem(ConstU32::<3>); }
ClearModp::from(ConstI32::<8>).test_mem(ConstU32::<3>);
```

On the SCALE target this is basically a no-op, on the Test target it does this comparison.

`x.test_value(y)`

Test whether the value held in x is the same as the value held in y. This is the same as `x.test()` except the value is compared to y and not the value emulated in the test environment.

5.4 SecretModp

This is the same as the sint in the underlying SCALE virtual machine.

5.4.1 Conversion of Data

SecretModp::from(a)

This takes an sint integer value a and loads it into a value of type **SecretModp**. You can also pass in an argument of a type **i64**, **ClearModp**, **SecretBit** and **SecretI64**.

```
let sa = SecretModp::from(ConstI32::<8>);

let aa: i64 = 10;
let mut a2=SecretModp::from(aa);

let cint_a = ClearModp::from(ConstI32::<10>);
let sint_a = SecretModp::from(cint_a);

let sb = SecretBit::from(false);
let ssb = SecretModp::from(sb);

// See the main SCALE manual for the semantics of this conversion
let si = SecretI64::from(ConstI32::<10>);
let ssi = SecretModp::from(si);
```

SecretModp::from_unsigned(a)

The above conversion from **SecretI64** to **SecretModp** is a signed conversion. To get an unsigned conversion you perform.

```
let si = SecretI64::from(ConstI32::<-10>);
let ssi = SecretModp::from_unsigned(si);
```

5.4.2 IO

The IO class functions for **SecretModp** datatypes can be access via the following commands:

SecretModp::private_input(p,c)

Loads an **SecretModp** from the channel c and player p.

```
let a=SecretModp::private_input(3, 10);
```

a.private_output(p,c)

Writes an **SecretModp** to channel c and player p.

```
let v = SecretModp::from(ConstI32::<1>);
v.private_output(0, 10);
```

5.4.3 Other Operations

`SecretModp::randomize()`

Produces a (secret) random number in the range $[0, \ldots, p-1]$, where p is the prime underlying the **SecretModp** type. This random number is the 'same' for all players.

```
let a = SecretModp::randomize();
```

`SecretModp::get_random_bit()`

Produces a (secret) random number in $\{0, 1\}$. This random bit is the 'same' for all players.

```
let a = SecretModp::get_random_bit();
```

`SecretModp::get_random_square()`

Produces a (secret) random number a in the range $[0, \ldots, p-1]$, where p is the prime underlying the **SecretModp** type, as well as its square b.

```
let (a,b) = SecretModp::get_random_square();
```

`SecretModp::get_random_triple()`

Produces two (secret) random numbers a and b in the range $[0, \ldots, p-1]$, where p is the prime underlying the **SecretModp** type, as well as their product c.

```
let (a,b,c) = SecretModp::get_random_triple();
```

5.4.4 Memory Access

`a.store_in_mem(x)`

To store data into memory location 50 say you need to execute, recalling that memory is accessed by 32 bit values.

```
unsafe{ sa.store_in_mem(ConstU32::<50>); }
```

To store to a variable location you use

```
let x: i64 = 50;
unsafe{ sa.store_in_mem(x); }
```

`SecretModp::load_from_mem(x)`

To load data from memory you do

```
let sa=SecretModp::load_from_mem(ConstU32::<50>);
let x: i64 = 50;
let sb=SecretModp::load_from_mem(x);
```

5.4.5 Testing Data

`x.test()`

In the SCALE target this takes the **SecretModp** value x, applies a reveal operation to it, and then writes the resulting **ClearModp** value into memory.

In the Test target this takes a value stored in the **ClearModp** memory saved on the last SCALE invocation, and compares it to x. If the two values are the same it prints the value, and the line number in the rust file where this was executed. Otherwise it aborts.

`x.test_value(y)`

Test whether the value held in x is the same as the value held in y. This is the same as `x.test()` except the value is compared to y and not the value emulated in the test environment.

5.5 SecretBit

This is the same as the sbit in the underlying SCALE virtual machine.

5.5.1 Conversion of Data

`SecretBit::from(a)`

This takes an integer value a and loads it into a value of type **SecretBit**.

```
let sa = SecretBit::from(true);
```

You can also load a **SecretModp** value into the secret bit, although it is up to the programmer to ensure that the value really is a bit. If it is not a bit, then some information can leak.

```
let cb = SecretModp::from(0);
let sa = SecretBit::from(sa);
```

5.5.2 Other Operations

`SecretBit::randomize()`

Produces a (secret) random bit. This random number is the 'same' for all players.

```
let a = SecretBit::randomize();
```

5.5.3 Memory Access

There is no **SecretBit** memory recall!

5.5.4 Testing Data

`x.test()`

In the SCALE target this takes the **SecretBit** value x, applies a reveal operation to it, and then writes the resulting regint value into memory.

In the Test target this takes a value stored in the regint memory saved on the last SCALE invocation, and compares it to x. If the two values are the same it prints the value, and the line number in the rust file where this was executed. Otherwise it aborts.

`x.test_value(y)`

Test whether the value held in x is the same as the value held in y. This is the same as `x.test()` except the value is compared to y and not the value emulated in the test environment.

5.6 Operators

5.6.1 Unary Operators

The following unary operations are allowed on **SecretBit**, **SecretI64**.

```
!
```

This operation inverts all bits (or just the one bit in the **SecretBit** case) of its argument. There is no ! operation like in C on integers. If you want to check whether all bits are zero in Rust, you need to use an equality operation.

5.6.2 Binary Operators

The following operations are allowed between **SecretI64** and **i64** types:

```
+    -    *    /
^    &    |
```

The output is of type **SecretI64** if any of the two operands are **SecretI64**s, otherwise the output type is a **i64**.

The following operations are allowed between **SecretModp** and **ClearModp** types:

```
+    -    *
```

The output is of type **SecretModp** if any of the two operands are **SecretModp**s, otherwise the output type is a **ClearModp**.

The following operations are allowed between **SecretModp** and **ClearModp** types:

```
/
```

The output is of type **SecretModp** if the first operand is **SecretModp**s, otherwise the output type is a **ClearModp**.
Note: The second operand must be a **ClearModp** or a `ConstI32` at present.

The following operations are allowed between **ClearModp** and **i64** types:

```
%    <<    >>    ^    &    |
```

The operations are performed by lifting the **ClearModp** values to the integers (in the range $[0, \ldots, p)$) and then performing the operation over the integers. The output is of type **ClearModp**

The following operations are allowed between **SecretBit** types:

```
^    &    |
```

The output is of type **SecretBit**

The following operations are allowed between **i64** types:

```
==      !=
<    >    <=   >=
```

The output is of type **i64/bool** and therefore allows branching on the resulting condition etc.

Between an **SecretI64**, **SecretModp** or **ClearModp** and an immediate operand (an `ConstU32`) you can execute shift-left and shift-right operations. These operators also exist if both operands are **i64**.

```
<<      >>
```

As in

```
let sb = sa << ConstU32::<3>;
```

For integers, the small example needs to use black_box as otherwise the optimizer will optimize the shift and just store the end result.

```
let a = black_box(2);
let b = black_box(1);
let c = a << b
```

6 Control Flow

In this section we detail the control flow operations which we support.

6.1 If-then-else

Branching on clear data is possible via the standard rust if-then-else construct. If c is a **i64** variable, then to branch on $c = 1$ you need to do

```
if c == 1 {
        ...
}
```

In contrast to Mamba, if you need to access data which has been altered within the if clauses you do not need to store these in memory. Instead you can just use the standard Rust local variables and everything will work as expected. For example

```
let c0 = black_box(0);

// Testing c0 = False
let a = if c0 == 1 {
    ClearModp::from(ConstI32::<0>)
} else {
    ClearModp::from(ConstI32::<25>)
};
// use 'a' from here on
```

6.2 While Loops

While loops can also be executed over **i64** datatypes, for example

```
let mut cond: i64 = 0;
while cond==0 {
    ... Do Something ...
    cond = ... some condition ...
}
```

If you want to access something inside the loop that can still be used outside the loop afterwards, use local variables similar to how the *cond* variable is used.

6.3 For Loops

Again standard rust loops can be executed, for example

```
let mut res = 1;
for i in 2..n  {
    res *= i;
}
a=a+res;
```

The compiler decides on heuristic optimization rules how and whether to unroll such loops. Whether the loop is unrolled or not has no effect on the correctness of the loop, but it can have effects on the efficiency of cryptographic operations.

6.4 Function Calls

Function calls can either be placed inline (which is really only a *hint* to the compiler) or they can be performed using stack-pushing of arguments, call/return and then popping outputs. The former corresponds to the Mamba 'Python-Function' behaviour, whilst the latter corresponds to the Mamba 'Mamba-Function' behaviour. However, with Rust if the compiler decides that the expanded version is too big it will revert to the call/return variant.

In the following `foo` is an unrolled function, whilst `bar` is a call/return style function.

```
#[inline(always)]
fn foo(x: i64) -> i64 {
    let mut y=1;
    for i in 2..x
        y *= i;
    y
}

#[inline(never)]
fn bar(x: i64) -> i64 {
    let mut y=1;
    for i in 2..x
        y *= i;
    y
}



#[inline(never)]
fn main() {
    let x = foo(10);
    let y = bar(10);
    ...
}
```

When using call/return the compiler works out all the stack pushing and popping for you. This is why usage of the stacks by the user might be unsafe, as the compiler really needs direct access to the stacks.

Function calls can also be recursive, unlike function calls in Mamba,

```
#[inline(never)]
fn fibonacci(x: u64) -> u64 {
    print!(x as i64, "\n");
    match x {
        0 => 0,
        1 | 2 => 1,
        _ => fibonacci(x - 1) + fibonacci(x - 2)
    }
}
```

Note that the compiler is much less likely to fully inline recursive functions unless it can tell when the recursion will end.

7 System Commands

This sections assume you kind of understand the SCALE system and have already programmed a bit in MAMBA. Eventually we will make it all self-contained. But for now lets assume you are an expert...

7.1 System Control Commands

start_clock(x)

Initializes the timer number x.

stop_clock(x)

Stops the timer number x.

```
start_clock(3);
....
stop_clock(3);
```

require_bit_length(x)

For some reason you program might require a minimum prime length. This command stores the maximum prime length which you signal, and then emits a single `REQBL` command at the beginning of the output assembler. Thus the Rust code

```
...
require_bit_length(10);
...
require_bit_length(20);
...
```

creates assembler with a single `REQBL 20` as the first instruction.

crash()

Crashes the system.

restart()

Executes the restart machinery.

clear_memory()

Executes a clear memory command.

clear_registers()

Executes a clear registers command.

open_channel(x)

Opens the communication channel to the outside world, returning the appropriate value as an **i64**.

```
let ans = open_channel(10_i32);
```

close_channel(x)

Closes the communication channel to the outside world.

```
close_channel(10_i32);
```

7.2 Stack Operations

Recall (from the main Scale documentation) there is a stack for all the five basic types **i64**, **SecretI64**, **ClearModp**, **SecretModp** and **SecretBit**. As the main Rust compiler also uses these stacks for function calls you need to be very careful when using the stacks. Thus you need to enclose them in an `unsafe` block. The basic methodology to use the stacks is the same for all five types, so we just give one example here.

```
let one = SecretI64::from(1);
let two = SecretI64::from(2);
let three = SecretI64::from(3);
let four = SecretI64::from(4);

unsafe {
    SecretI64::push(one);
    SecretI64::push(two);
    let sp = SecretI64::get_stack_pointer();
    let c = SecretI64::peek(sp - 1);          // Peek relative to pos 0
    let cc = SecretI64::peek_from_top(1);     // Peek relative to pos stack_ptr
    SecretI64::poke(sp, &three);              // Poke relative to pos 0
    SecretI64::poke_from_top(1, &four);       // Poke relative to pos stack_ptr
    let d = SecretI64::pop();
    let e = SecretI64::pop();
}
```

7.3 Printing

Access to the printing commands is performed via the `print!` command. Currently, only strings and **ClearModp** types can be output in this way.

```
let ca = ClearModp::from(1);
let cb = ClearModp::from(2);
print!("hello ", ca, " world ", cb, "\n");
```

If you cannot be bothered to add the newline at the end you can also use `println!`

```
let ca = ClearModp::from(1);
let cb = ClearModp::from(2);
println!("hello ", ca, " world ", cb);
```

7.4 Circuits and Local Functions

For details of how these work in general see the main SCALE manual. If you want to add your own circuits/local functions see the files

- `WebAssembly/scale_std/src/circuits.rs`
- `WebAssembly/scale_std/src/local_functions.rs`

which should be relatively self-explanatory as to how to add new functions to the underlying system.

7.4.1 Inbuilt Circuits

The inbuilt circuits shipped with SCALE can be called using the following function signatures (the ones associated to floating point operations can be accessed via the `SecretIEEE` type).

```
pub fn AES128( key128: [SecretI64; 2], mess: [SecretI64; 2])
              -> [SecretI64; 2]

pub fn AES192( key128: [SecretI64; 3], mess: [SecretI64; 2])
              -> [SecretI64; 2]

pub fn AES256( key128: [SecretI64; 4], mess: [SecretI64; 2])
              -> [SecretI64; 2]

pub fn SHA3( istate: Array<SecretI64, 25>) -> Array<SecretI64, 25>

pub fn SHA256( mess: Array<SecretI64, 8>, state: Array<SecretI64, 4>)
              -> Array<SecretI64, 4>

pub fn SHA512( mess: Array<SecretI64, 16>, state: Array<SecretI64, 8>)
              -> Array<SecretI64, 8>

pub fn IEEE_add( input: [SecretI64; 2] ) -> SecretI64

pub fn IEEE_mul( input: [SecretI64; 2] ) -> SecretI64

pub fn IEEE_div( input: [SecretI64; 2] ) -> SecretI64

pub fn IEEE_eq( input: [SecretI64; 2] ) -> SecretI64

pub fn IEEE_f2i( input: SecretI64 ) -> SecretI64

pub fn IEEE_i2f( input: SecretI64 ) -> SecretI64

pub fn IEEE_sqrt( input: SecretI64 ) -> SecretI64

pub fn IEEE_lt( input: [SecretI64; 2] ) -> SecretI64
```

For example, to call the AES-128 circuit you would execute:

```
#[inline(always)]
fn main() {

    let zero = SecretI64::from(0);
    let one = SecretI64::from(1);
    let mone = SecretI64::from(-1);

    let key128: [SecretI64; 2] = [zero, mone];
    let mess: [SecretI64; 2] = [mone, one];

    let ciph = AES128(key128,mess);
}
```

7.4.2 Inbuilt Local Functions

The calling of Local Functions works in much the same way. If you look at the implementation of matrix multiplication of two matrices[1] of type **ClearModp** in `WebAssembly/scale_std/src/local_functions.rs` you will find the implementation:

```
const C_MULT_C: u32 = 0;

#[inline(always)]
#[allow(non_snake_case)]
pub fn Matrix_Mul_CC<const N: u64, const L: u64, const M: u64>
                    ( A: &Matrix::<ClearModp, N, L>,
                      B: &Matrix::<ClearModp, L, M>
                    ) -> Matrix::<ClearModp, N, M> {
    let (col,row,C)= unsafe { execute_local_function!(C_MULT_C(
        N as i64,
        L as i64,
        *A,
        L as i64,
        M as i64,
        *B
    ) ->
        i64,
        i64,
        Matrix::<ClearModp, N, M>
    ) };
    if row != (N as i64) || col != (M as i64) {
        crash();
    }
    C
}
```

We explain this now, as you can replicate this to add your own Local Functions:

- The 'unsafe' part calls the Local Function with index zero.
- The arguments are pushed onto the stack in the order N, L, A, L, M and B.
- The operation is then called, with the results popped off the stack in the order col, row and C.
- After some doubly checking the matrix C is returned to the caller.

The caller code is given by

```
let mut A = Matrix::<ClearModp, N, L>::uninitialized();
let mut B = Matrix::<ClearModp, L, M>::uninitialized();

... Set data in A and B ...

let C = Matrix_Mul_CC(&A, &B);
```

Currently there are four Local Functions defined, with the signatures (the ones in the main SCALE manual associated to floating point operations can be accessed via the `ClearIEEE` type directly).

[1] See later for the matrix types

```
pub fn Matrix_Mul_CC<const N: u64, const L: u64, const M: u64>
                ( A: &Matrix::<ClearModp, N, L>,
                  B: &Matrix::<ClearModp, L, M>
                ) -> Matrix::<ClearModp, N, M>

pub fn Matrix_Mul_SC<const N: u64, const L: u64, const M: u64>
                ( A: &Matrix::<SecretModp, N, L>,
                  B: &Matrix::<ClearModp, L, M>
                ) -> Matrix::<SecretModp, N, M>

pub fn Matrix_Mul_CS<const N: u64, const L: u64, const M: u64>
                ( A: &Matrix::<ClearModp, N, L>,
                  B: &Matrix::<SecretModp, L, M>
                ) -> Matrix::<SecretModp, N, M>

pub fn Gauss_Elim<const N: u64, const M: u64>
              ( A: &Matrix::<ClearModp, N, M>,
              ) -> Matrix::<ClearModp, N, M>
```

8 Standard Library

8.1 Boxes

A box is essentially an array of size one. It is a 'safe' way for users to write/read from the SCALE memory one element at a time. To load the standard library version of a Box use

```
use scale_std::heap::Box;
```

As it is tied to the SCALE memory it only comes in the types **i64**, **SecretI64**, **ClearModp** and **SecretModp**.

`Box::uninitialized()`

To initialise a box with no pre-defined value use

```
let mut a= Box<SecretI64>::uninitialized();
```

`a.set(x)`

Does the obvious.

`a.get()`

Again does the obvious.

8.2 Arrays

This data type is implemented using the SCALE memory. It is currently available for the following types; **i64**, **SecretI64**, **ClearModp**, **SecretModp**, `ClearFixed<K,F>`, `SecretFixed<K,F>`, `ClearFloat<V,P>`, `SecretFloat<V,P>`, `ClearIEEE` and `SecretIEEE`. You can always use simple arrays of the form

```
let key128: [SecretI64; 2] = [zero, none];
```

as in the earlier AES example. But these will not compile as soon as the size becomes too large. The simple arrays will compile to registers, and thus be more efficient than arrays stored in memory. But arrays stores in memory are more flexible. To load the standard library version of Arrays use

```
use scale_std::array::Array;
```

`Array::uninitialized()`

To initialise an array with no pre-defined values use

```
let mut a= Array<SecretI64, 25>::uninitialized();
```

`a.fill(x)`

Fills an array, used with initialization

```
let a: Array<SecretI64, 25> = Array::fill(zero);
```

`a.set(i,x)`

Does the obvious, but has no bound checks, i.e. you can access the 50th element of a 25 element array.

`a.get(i)`

Get the reference of an element from the array by checking we do not make any memory overflow

a.get_unchecked(i)

Get the reference of an element from the array without checking we do not make any memory overflow

a.get_mut(i)

Get the mutable reference of an element from the array by checking we do not make any memory overflow.

a.get_mut_unchecked(i)

Get the mutable reference of an element from the array without checking we do not make any memory overflow

drop(a)

Deletes the array from memory.

a.iter()

Iterator over an `Array`,

```
for (i, val) in a.iter().enumerate() {
    ....
}
```

To iterate backwards use

```
for (i, val) in a.iter().rev().enumerate() {
    ....
}
```

For more details on using iterators see `https://doc.rust-lang.org/std/iter/trait.Iterator.html`.

a.addr(i)

This returns the address in memory of the i-th element in the array.

a.reverse()

Returns the same array, but with the elements in the reverse order, i.e. $a[i] = a[\ell - 1 - i]$ for $i = 0, \ldots, \ell - 1$.

a.reveal()

For the secret types this creates a non-secret version, as in.

```
let cint_array = sint_array.reveal();
```

a.slice(range)

Converts (part of) an array to a slice, of which the length is only known at runtime.

```
let a_slice: Slice<SecretI64> = a.slice(3..);
```

8.2.1 Stack Operations on Arrays

One can also apply the stack operations on an array, which applies to all the elements in the array. Note the pop operation works in reverse, so that pushing and then popping gives the same array. The stack read operations (pop, peek etc) will pop and peek the entire array, so you need to ensure the stack really has that many elements on it; i.e. no access is performed outside the stack, otherwise a runtime crash will occur.

push(x), pop()

peek(i), poke(i, x)

peek_from_top(i), poke_from_top(i, x)

These are called using the following convention

```
unsafe { Array::push(&a) }
let a = unsafe { Array::<SecretI64, 25>::pop() };
```

8.3 Slices

This data type is equal to Arrays, the only difference being that the length of a slice is not known at compile time. Just like Arrays, this data type is implemented using the SCALE memory. It is currently available for the following types **i64**, **SecretI64**, **ClearModp**, **SecretModp**, `ClearFixed<K,F>`, `SecretFixed<K,F>`, `ClearFloat<V,P>`, `SecretFloat<V,P>`, `ClearIEEE` and `SecretIEEE` data types. To load the standard library version of slices use

```
use scale_std::slice::Slice;
```

Slice::uninitialized()

To initialise a slice with no pre-defined values use

```
let mut a= Slice<SecretI64>::uninitialized();
```

s.set(i,x)

Does the obvious.

a.get(i)

Get the reference of an element from the slice by checking we do not make any memory overflow

a.get_unchecked(i)

Get the reference of an element from the slice without checking we do not make any memory overflow

a.get_mut(i)

Get the mutable reference of an element from the slice by checking we do not make any memory overflow.

a.get_mut_unchecked(i)

Get the mutable reference of an element from the slice without checking we do not make any memory overflow

drop(a)

Deletes the slice from memory.

s.iter()

Iterator over an `Slice`,

```
for (i, val) in s.iter().enumerate() {
    ....
}
```

To iterate backwards use

```
for (i, val) in a.iter().rev().enumerate() {
    ....
}
```

s.addr(i)

This returns the address in memory of the i-th element in the slice.

s.reverse()

Returns the same slice, but with the elements in the reverse order, i.e. $s[i] = s[\ell - 1 - i]$ for $i = 0, \ldots, \ell - 1$.

s.reveal()

For the secret types this creates a non-secret version, as in.

```
let cint_slice = sint_slice.reveal();
```

s.slice(range)

Returns part of the original slice.

```
let s_slice: Slice<SecretI64> = s.slice(3..);
```

s.len()

Returns the length of the slice as a u64-number.

```
let length: u64 = s.len();
```

8.4 Arithmetic on Arrays and Slices

These operations only apply to special types of Arrays or Slices.

8.4.1 Private IO

Since private IO requires rounds of communication we provide helper functions for Arrays and Slices which use a constant number of rounds, irrespective of the size of the Array or Slice.

Array<SecretModp,N>::private_input(p,c)

a.private_output(p,c)

Slice<SecretModp>::private_input(N, p,c)

s.private_output(p,c)

These are all called in the following manner

```
let a: Array<SecretModp, 5>=Array::private_input(1, 10);
a.private_output(2, 10);


let b: Slice<SecretModp>=Slice::private_input(6, 1, 10);
b.private_output(2, 10);
```

8.4.2 Addition/Subtraction/Multiplication

Vectorized addition, subtraction, multiplication, division and modular-remainder of Arrays and Slices of type **ClearModp** and **SecretModp** are implemented via either an operator notation, or a member function notation. These are not implemented for secret-secret multiplication and the division/modular-remainder are only implemented for clear-clear operations. The operator variants are often slower, as they can require a `clone` before calling (depending on the usage of the arguments in other parts of the code). The member function variants never require a clone. We present an example for Arrays, an equivalent syntax applies to Slices,

```
const N: u64=200;
let mut ca_arr: Array<ClearModp, N> = Array::uninitialized();
let mut cb_arr: Array<ClearModp, N> = Array::uninitialized();
let mut sa_arr: Array<SecretModp, N> = Array::uninitialized();
let mut sb_arr: Array<SecretModp, N> = Array::uninitialized();

...

// Operator notation (V slow)
let a_add_cc = ca_arr.clone() + cb_arr.clone();
let a_add_cs = ca_arr.clone() + sb_arr.clone();
let a_add_sc = sa_arr.clone() + cb_arr.clone();
let a_add_ss = sa_arr.clone() + sb_arr.clone();
let a_sub_cc = ca_arr.clone() - cb_arr.clone();
let a_sub_cs = ca_arr.clone() - sb_arr.clone();
let a_sub_sc = sa_arr.clone() - cb_arr.clone();
let a_sub_ss = sa_arr.clone() - sb_arr.clone();
let a_mul_cc = ca_arr.clone() * cb_arr.clone();
let a_mul_cs = ca_arr.clone() * sb_arr.clone();
let a_mul_sc = sa_arr.clone() * cb_arr.clone();
let a_div_cc = ca_arr.clone() / cb_arr.clone();
let a_mod_cc = ca_arr.clone() % cb_arr.clone();


// Operator notation (slow)
let a_add_cc = &ca_arr + cb_arr.clone();
...
let a_mod_cc = &ca_arr % cb_arr.clone();
```

```
// Operator notation (slow)
let a_add_cc = ca_arr.clone() + &cb_arr;
...
let a_mod_cc = ca_arr.clone() % &cb_arr;


// Operator notation (fastest)
let a_add_cc = &ca_arr + &cb_arr;
...
let a_mod_cc = &ca_arr % &cb_arr;


// Member function notation (fastest)
let a_add_cc_t = ca_arr.add_clear(&cb_arr);
let a_add_cs_t = ca_arr.add_secret(&sb_arr);
let a_add_sc_t = sa_arr.add_clear(&cb_arr);
let a_add_ss_t = sa_arr.add_secret(&sb_arr);
let a_sub_cc_t = ca_arr.sub_clear(&cb_arr);
let a_sub_cs_t = ca_arr.sub_secret(&sb_arr);
let a_sub_sc_t = sa_arr.sub_clear(&cb_arr);
let a_sub_ss_t = sa_arr.sub_secret(&sb_arr);
let a_mul_cc_t = ca_arr.mul_clear(&cb_arr);
let a_mul_cs_t = ca_arr.mul_secret(&sb_arr);
let a_mul_sc_t = sa_arr.mul_clear(&cb_arr);
let a_div_cc_t = ca_arr.div_clear(&cb_arr);
let a_mod_cc_t = ca_arr.mod_clear(&cb_arr);
```

The slice variants do not check whether the two source arrays have the same number of elements, the size is always taken to be that of `*this`.

8.4.3 Advanced Arithmetic on Arrays and Slices

`a.evaluate(c)`

Given an array or slice a of length ℓ of type **ClearModp** or **SecretModp** this evaluates the polynomial $\sum_{i=0}^{\ell-1} a_i \cdot c^i$ for the **ClearModp** value c.

`Array<ClearModp,N>::bit_decomposition_ClearModp(c)`

`Slice<ClearModp>::bit_decomposition_ClearModp(c, N)`

`Array<i64,N>::bit_decomposition_i64(v)`

`Slice<i64>::bit_decomposition_i64(v, N)`

Given a **ClearModp** value c or an **i64** value v, these functions compute the bit-decomposition of c (resp. v) upto the N-th bit position. Returning the result as an Array or Slice.

Array<ClearModp,N>::bit_decomposition_ClearModp_Signed(c)

Slice<ClearModp>::bit_decomposition_ClearModp_Signed(c, N)

The same but assumes the value c is signed, i.e. if $p > 2$ then we bit-decompose the negative value $c - p$.

8.5 Matrices

Again this data type is implemented using the SCALE memory. It is thus only available for **i64**, **SecretI64**, **ClearModp** and **SecretModp** data types. To load the standard library version of Matrices use

```
use scale_std::matrix::Matrix;
```

There are a similar set of functions as for the Array type.

Matrix::unitialized()

To initialise a matrix with no pre-defined values, with fives rows and four columns, use

```
let mut a= Matrix<SecretI64, 5, 4>::unitialized();
```

a.fill(x)

Fills an array, used with initialization

```
let a: Matrix<SecretI64, 5, 4> = Matrix::fill(zero);
```

a.set(i,j,x)

Does the obvious.

a.get(i,j)

Again does the obvious

a.get_row(i)

Does the obvious, returning the result as an `Array`.

a.get_column(i)

Again does the obvious

a.iter()

Iterator over a `Matrix`,

```
for (i, val) in a.iter().enumerate() {
    ....
}
```

The iterator works in the order $a_{0,0}, a_{0,1}, a_{0,2}, \ldots, a_{0,c-1}, a_{1,0}, a_{1,1}, \ldots, a_{r-1,c-1}$.
To iterate backwards use

```
for (i, val) in a.iter().rev().enumerate() {
    ....
}
```

a.reveal()

For the secret types this creates a non-secret version, as in.

```
let cint_matrix = sint_matrix.reveal();
```

8.5.1 Stack Operations on Matrices

One can also apply the stack operations on a matrix, which applies to all the elements in the matrix. The push operation works in the same order as the above iterator, with the pop operations working in reverse. The stack read operations (pop, peek etc) will pop and peek the entire matrix, so you need to ensure the stack really has that many elements on it; i.e. no access is performed outside the stack, otherwise a runtime crash will occur.

push(x), pop()

peek(i), poke(i, x)

peek_from_top(i), poke_from_top(i, x)

These are called using the following convention

```
unsafe { Matrix::push(&a) }
let a = unsafe { Matrix::<SecretI64, 5, 4>::pop() };
```

8.6 ClearIEEE

This class gives direct access to the IEEE-754 compliant implementation of floating point numbers in SCALE. The operations on `ClearIEEE` values are usually much faster than processing using `ClearFixed` values, as the basic arithmetic is implemented by passing the operation (via a local function) to the SCALE engine; where it is implemented in C++. The usage is demonstrated in the example below:

```
use scale_std::ieee::*;
#[inline(always)]
#[scale::main(KAPPA = 40)]
fn main() {
  let c= ClearIEEE::from(3.141592);
  print!(c,"\n");
}
```

ClearIEEE::from(x)

You can convert an **i64**, a `ClearFixed<K,F>` or a constant `f64` to a `ClearIEEE` as follows:

```
let c= ClearIEEE::from(3.141592);

let i: i64= 1677216;
let fi=ClearIEEE::from(i);
```

You can also convert a `ClearIEEE` to an **i64**, which rounds the floating point value to a 64-bit integer (if possible).

```
let ic=i64::from(c);
```

a.rep()

The internal representation as an **i64** can be obtained by the `rep()` function.

a.set(x)

One can set the internal representation to a **i64** value using the `set()` function.

ClearIEEE::NaN()

Returns a representation of IEEE NaN.

ClearIEEE::randomize()

Produces a random value in the range $[0, \ldots, 1)$. This random number is the 'same' for all players.

```
let a = ClearIEEE::randomize();
```

8.6.1 Comparisons

The `ClearIEEE` type implements the `Eq`, `PartialEq` and `PartialOrd` traits. Therefore you can apply normal comparison operators to `ClearIEEE` types as in...

```
fn test_approx(a: ClearIEEE, val: ClearIEEE) -> i64
{
    let lower = val - ClearIEEE::from(0.00001);
    let upper = val + ClearIEEE::from(0.00001);
    let ok = (a>lower) | (a<upper);
    ok as i64
}
```

To aid with generic programming the following member functions are also implemented, which output a value of type **i64**,

a.gt(x), a.ge(x), a.lt(x), a.le(x), a.gt(x), a.eq(x), a.ne(x)

a.gtz(), a.gez(), a.ltz(), a.lez(), a.gtz(), a.eqz(), a.nez(x)

8.7 SecretIEEE

This class gives direct access to the IEEE-754 compliant implementation of floating point numbers in SCALE. Recall these are implemented using evaluations of binary circuits, so are not as fast as the `sfloat` types implemented in MAMBA (which are not currently available in the Rust interface, but we hope to fix this soon). The `SecretFixed` types is more efficient than the `SecretIEEE` type and should usually be preferred. The benefit of the `SecretIEEE` type is that it is more expressive in terms of the numbers ranges it can hold; but this comes at the drawback of significant reduced performance.

SecretIEEE::from(x)

You can convert a **SecretI64**, a `ClearIEEE` or a constant `f64`to a `SecretIEEE` as follows:

```
let c= ClearIEEE::from(3.141592);
let sc=SecretIEEE::from(c);

let i: i64= 1677216;
let si= SecretI64::from(i);
let sfi=SecretIEEE::from(si);
```

You can also convert a `SecretIEEE` to a **SecretI64**, which rounds the floating point value to a 64-bit integer (if possible).

```
let ic=SecretI64::from(sc);
```

a.rep()

The internal representation as a **SecretI64** can be obtained by the `rep()` function.

a.set(x)

One can set the internal representation to a **SecretI64** value using the `set()` function.

s.reveal()

For the `SecretIIEEE` type this creates a `ClearIEEE` version, as in.

```
let ci = si.reveal();
```

SecretIEEE::NaN()

Returns a representation of IEEE NaN.

SecretIEEE::randomize()

Produces a (secret) random value in the range $[0, \ldots, 1)$. This random number is the 'same' for all players.

```
let a = SecretIEEE::randomize();
```

8.7.1 Comparisons

The following 'operators' can be applied between two `SecretIEEE` values The output is a **SecretBit** value. As the result of the operator cannot be used in a conditional branch, we use the member function notation for such 'operators', as opposed to the operator notation. Thus syntactically the programmer is less likely to make a mistake.

a.gt(x), a.ge(x), a.lt(x), a.le(x), a.gt(x), a.eq(x), a.ne(x)

The following give variants which compare just to zero.

a.gtz(), a.gez(), a.ltz(), a.lez(), a.gtz(), a.eqz(), a.nez(x)

8.8 Arithmetic of ClearIEEE and SecretIEEE

The following operations are allowed between two elements of type `SecretIEEE` and `ClearIEEE`.

```
+   -   *   /
```

Clear and secret values can be combined to result in a secret value.

8.9 Bit Operations

There are a number of routines to access and process Slices of bits, which are represented in either **ClearModp** or **SecretModp** format.

BitAdd(Slice<U> A, Slice<V> B)

This produces a slice of **SecretModp** values corresponding to the summation of the bits in A and B. The two input slices are assumed to be of the same length k, and the output slice is of length $k+1$. The type of U and V can be either **SecretModp** or **ClearModp**, bit not both **ClearModp**.

```
// 7 + 5 = 12
let mut ca: Slice<ClearModp> = Slice::uninitialized(3);
ca.set(0,&ClearModp::from(1));
ca.set(1,&ClearModp::from(1));
ca.set(2,&ClearModp::from(1));

let mut sb: Slice<SecretModp> = Slice::uninitialized(3);
sb.set(0,&SecretModp::from(1));
sb.set(1,&SecretModp::from(0));
sb.set(2,&SecretModp::from(1));

let v_ca_sb = BitAdd(&ca,&sb);
```

BitIncrement(Array<SecretModp, K> A)

On input of an array of bits, represented as **SecretModp** values, this function returns the slice of **SecretModp** values consting of the incremeneted integer value.

```
let mut sa: Array<SecretModp, 3> = Array::uninitialized();
sa.set(0,&SecretModp::from(1));
sa.set(1,&SecretModp::from(1));
sa.set(2,&SecretModp::from(1));
let v_inc_sa = BitIncrement(&sa);
```

BitLT(a, B, K)

This takes a **ClearModp** value B and an Slice of **SecretModp** values A, representing bits (of size at least K). It computes the **SecretModp** value representing the conditional $a < \sum_{i=0}^{K-1} b_i \cdot 2^i$.

```
let mut saa: Slice<SecretModp> = Slice::uninitialized(3);
saa.set(0,&SecretModp::from(1));
saa.set(1,&SecretModp::from(0));
saa.set(2,&SecretModp::from(1));

let four = ClearModp::from(4);
let six = ClearModp::from(6);
let cmp_four = BitLT(four, saa, 3);
let cmp_six = BitLT(six, saa, 3);
```

BitLTFull(Slice<U> A, Slice<V> B))

The two input slices are assumed to be of the same length K, and the output is a single **SecretModp** value representing the conditional $\sum_{i=0}^{K-1} a_i \cdot 2^i < \sum_{i=0}^{K-1} b_i \cdot 2^i$. The type of U and V can be either **SecretModp** or **ClearModp**, bit not both **ClearModp**. This routine does not utilize any statistical security gap, thus K can be the size of $\log_2 p$.

```
let mut ca: Slice<ClearModp> = Slice::uninitialized(3);
ca.set(0,&ClearModp::from(1));
ca.set(1,&ClearModp::from(1));
ca.set(2,&ClearModp::from(1));

let mut sb: Slice<SecretModp> = Slice::uninitialized(3);
sb.set(0,&SecretModp::from(1));
sb.set(1,&SecretModp::from(0));
sb.set(2,&SecretModp::from(1));

let cmp_a1 = BitLTFull(&ca, &sb);
```

BitDec::<K,M,KAPPA>(s)

Given a (signed) **SecretModp** value which lies in $\mathbb{Z}_{\langle k \rangle}$ this produces the first M bits of the bit-decomposition using statistical security parameter `KAPPA`. The output is given as a slice of **SecretModp** values, each of which represents a bit.

```
let ss1 = SecretModp::from(5);
let ss2 = SecretModp::from(-5);
let v1 = BitDec::<10,5,40>(ss1);
let v2 = BitDec::<10,5,40>(ss2);
```

bitdec::<KAPPA>(s, k)

Same but now k can be a run time variable and not a compile time constant, and we use $K = M = k$.

```
let ss1 = SecretModp::from(5);
let ss2 = SecretModp::from(-5);
let v1 = bitdec::<40>(ss1, 10);
let v2 = bitdec::<40>(ss2, 10);
```

BitDecFull(x)

Given a **SecretModp** value this produces the bit-decomposition into a slice of dimension $\log_2 p$. The value x can be in the range $[0, \ldots, p)$ and no usage is made of any statistical security parameter. The output is given as a slice of **SecretModp** values, each of which represents a bit.

```
let ss1 = SecretModp::from(5);
let ss2 = SecretModp::from(-5);
let v1 = BitDecFull(ss1);
let v2 = BitDecFull(ss2);
```

8.10 ClearInteger

We define $\mathbb{Z}_{\langle k \rangle}$ as the set of integers $\{x \in \mathbb{Z} : -2^{k-1} \leq x \leq 2^{k-1} - 1\}$, which we embed into $\mathbb{F}_p$ via the map $x \mapsto x \pmod p$. In MAMBA the user had to manually think of elements in $\mathbb{F}_p$ as representing values in $\mathbb{Z}_{\langle k \rangle}$, and manually keep track of their sizes for all the 'advanced' integer operations. This was efficient, but prone to errors and misunderstandings. In Rust we have a seperate type to represent elements in $\mathbb{Z}_{\langle k \rangle}$, both in clear and secret form. This is strongly typed in the sense that a `ClearInteger<3>` is a different type from a `ClearInteger<4>`.

This class enables one to load and print clear integers, and acts mainly as a helper class for the `SecretInteger` class below. The usage is demonstrated in the example below:

```
use scale_std::integer::*;
#[inline(always)]
#[scale::main(KAPPA = 40)]
fn main() {
  let c: ClearInteger<3>= ClearInteger::from(2 as i64);
  print!(c,"\n");
}
```

ClearInteger::from(x)

You can convert a **ClearModp** or a **i64** to a `ClearInteger<K>` as follows:

```
let c = ClearModp::from(2);
let ci: ClearInteger<3>= ClearInteger::from(c);
let ci1: ClearInteger<3>= ClearInteger::from(2 as i64);
```

a.recast()

A value of one size can be recast to another using the `recast` member function. It is assumed the user knows what they are doing here and that such recasting will not necessarily result in a valid representation being created.

```
let c = ClearModp::from(2);
let ci: ClearInteger<3>= ClearInteger::from(c);
let ci1: ClearInteger<5>= unsafe{ a.recast() };
```

a.rep()

Returns the internal representation as a `ClearModp` value.

a.set(v)

Sets the internal representation to be the `ClearModp` value v.

a.Trunc(M,S)

Implements the deterministic trunction operation of rounding $a/2^M$. When $S = \mathsf{true}$ the value a is assumed to lie in $\mathbb{Z}_{\langle k \rangle}$, but when $S = \mathsf{false}$ the value a is assumed to lie in $[0, \ldots, 2^{K-1})$.

```
let a: ClearInteger::<6> = ClearInteger::from(23);
let b = a.Trunc(3, true);
```

a.Mod2m(M,S)

This computes the value of a modulo 2^M. When $S = \mathsf{true}$ the value a is assumed to lie in $\mathbb{Z}_{\langle k \rangle}$, but when $S = \mathsf{false}$ the value a is assumed to lie in $[0, \ldots, 2^{K-1})$. Mathematically this computes the expression

$$\left(a + S \cdot 2^{K-1}\right) \pmod 2^M.$$

Implicitly this assumes that $M < K$.

```
let a: ClearInteger<10> = ClearInteger::from(-24);
let b = a.Mod2m(5, true);
```

a.Mod2(S)

As above but takes $M = 1$.

ClearInteger::randomize()

Produces a random value in $\mathbb{Z}_{\langle k \rangle}$. This random number is the 'same' for all players.

```
let a: ClearInteger<20> = ClearInteger::randomize();
```

8.10.1 Comparisons

The following 'operators' can be applied between two `ClearInteger<K>` values,the output being a **ClearModp** value. As the result of the operator cannot be used in a conditional branch, we use the member function notation for such 'operators', as opposed to the operator notation. Thus syntactically the programmer is less likely to make a mistake.

a.gt(x), a.ge(x), a.lt(x), a.le(x), a.gt(x), a.eq(x), a.ne(x)

The following give variants which compare just to zero.

a.gtz(), a.gez(), a.ltz(), a.lez(), a.gtz(), a.eqz(), a.nez(x)

8.11 SecretInteger

Many operations on `SecretIntegers` make use of the statistical security gap `KAPPA` defined in the pre-amble.

SecretInteger::from(x)

You can convert a **ClearModp** or a `ClearInteger<K>` to a `SecretInteger<K>` as follows:

```
let c = ClearModp::from(2);
let ci: ClearInteger<3>= ClearInteger::from(c);
let si1: SecretInteger<3>= SecretInteger::from(c);
let si2: SecretInteger<3>= SecretInteger::from(ci);
```

a.recast()

A value of one size can be recast to another using the `recast` member function. It is assumed the user knows what they are doing here and that such recasting will not necessarily result in a valid representation being created.

```
let c = ClearModp::from(2);
let si: SecretInteger<3>= SecretInteger::from(c);
let si1: SecretInteger<5>= unsafe{ a.recast() };
```

a.rep()

Returns the internal representation as a `SecretModp` value.

a.set(v)

Sets the internal representation to be the `SecretModp` value v.

a.reveal()

For the `SecretInteger` typs this creates a `ClearInteger` version, as in.

```
let ci = si.reveal();
```

a.Mod2m(M,S)

This computes the value of a modulo 2^M. When $S = \mathsf{true}$ the value a is assumed to lie in $\mathbb{Z}_{\langle k \rangle}$, but when $S = \mathsf{false}$ the value a is assumed to lie in $[0, \ldots, 2^{K-1})$. Mathematically this computes the expression

$$\left(a + S \cdot 2^{K-1}\right) \pmod 2^M.$$

Implicitly this assumes that $M < K$.

```
let a: SecretInteger<10> = SecretInteger::from(-24);
let b = a.Mod2m(5, true);
```

a.Mod2(S)

As above but takes $M = 1$.

Pow2::<K,KAPPA>(b)

If b is a **SecretModp** in the range $[0, \ldots, K)$ this computes the value 2^b as a `SecretInteger<K>`

```
let b = SecretModp::from(28);
let d = Pow2::<32,40>(b);
```

a.TruncPr(M,S)

Implements the probabilistic trunction operation of rounding $a/2^M$. When $S = \mathsf{true}$ the value a is assumed to lie in $\mathbb{Z}_{\langle k \rangle}$, but when $S = \mathsf{false}$ the value a is assumed to lie in $[0, \ldots, 2^{K-1})$.

```
let a: SecretInteger::<6> = SecretInteger::from(23);
let b = a.TruncPr(3, true);
```

a.Trunc(M,S)

Implements the deterministic trunction operation of rounding $a/2^M$. When $S = \mathsf{true}$ the value a is assumed to lie in $\mathbb{Z}_{\langle k \rangle}$, but when $S = \mathsf{false}$ the value a is assumed to lie in $[0, \ldots, 2^{K-1})$.

```
let a: SecretInteger::<6> = SecretInteger::from(23);
let b = a.Trunc(3, true);
```

a.TruncRoundNearest(M,S)

Implements the deterministic rounding to the nearest integer operation of rounding $\lceil a/2^M \rfloor$. When $S = \mathsf{true}$ the value a is assumed to lie in $\mathbb{Z}_{\langle k \rangle}$, but when $S = \mathsf{false}$ the value a is assumed to lie in $[0, \ldots, 2^{K-1})$.

```
let a: SecretInteger::<6> = SecretInteger::from(23);
let b = a.TruncRoundNearest(3, true);
```

a.ObliviousTrunc(m)

Implements `Trunc` operation where m is a **SecretModp** value which lies in the range $[0, \ldots, K)$. The output is an `Array` of three **SecretModp** values. The first element is $t = \lfloor a/2^m \rfloor$, the second is $a - 2^m \cdot t$ and the third element is 2^m.

```
let a = SecretModp::from(23);
let b: SecretInteger::<6> = SecretInteger::from(a);
let m = SecretModp::from(3);
let c = b.ObliviousTrunc(m);
```

SecretInteger::randomize()

Produces a (secret) random value in $\mathbb{Z}_{\langle k \rangle}$. This random number is the 'same' for all players.

```
let a: SecretInteger<20> = SecretInteger::randomize();
```

8.11.1 Comparisons

The following 'operators' can be applied between two `SecretInteger<K>` values,the output being a **SecretModp** value. As the result of the operator cannot be used in a conditional branch, we use the member function notation for such 'operators', as opposed to the operator notation. Thus syntactically the programmer is less likely to make a mistake.

a.gt(x), a.ge(x), a.lt(x), a.le(x), a.gt(x), a.eq(x), a.ne(x)

The following give variants which compare just to zero.

a.gtz(), a.gez(), a.ltz(), a.lez(), a.gtz(), a.eqz(), a.nez(x)

8.12 Arithmetic of ClearInteger and SecretInteger

Since a reduction operation (to ensure the value really lies in $\mathbb{Z}_{\langle k \rangle}$) is expensive we give function and operator versions of arithmetic. Warning: *In the member function versions no reduction occurs, thus these can be used when the user knows that no wrap around will occur. We thus mark them as* **unsafe**. The operator versions allow for ease of use, and guarantee correctness, but will be slower than the function versions.

The operators unarray minus (i.e. negation), addition, subtraction and multiplication are supported. If a clear and secret value are combined then the result is secret. The equivalent function versions for arithmetic have the following signatures. Note all are **unsafe** bar the `negate` function.

```
ClearInteger<K>::negate()                          -> ClearInteger<K>;
ClearInteger<K>::add(ClearInteger<K>)              -> ClearInteger<K>;
ClearInteger<K>::sub(ClearInteger<K>)              -> ClearInteger<K>;
ClearInteger<K>::mul(ClearInteger<K>)              -> ClearInteger<K>;
ClearInteger<K>::add_secret(SecretInteger<K>)      -> SecretInteger<K>;
ClearInteger<K>::sub_secret(SecretClearInteger<K>) -> SecretInteger<K>;
ClearInteger<K>::mul_secret(SecretInteger<K>)      -> SecretInteger<K>;
SecretInteger<K>::negate()                         -> SecretInteger<K>;
SecretInteger<K>::add(SecretInteger<K>)            -> SecretInteger<K>;
SecretInteger<K>::sub(SecretInteger<K>)            -> SecretInteger<K>;
SecretInteger<K>::mul(SecretInteger<K>)            -> SecretInteger<K>;
```

```
SecretInteger<K>::add_clear(ClearInteger<K>)      -> SecretInteger<K>;
SecretInteger<K>::sub_clear(ClearInteger<K>)      -> SecretInteger<K>;
SecretInteger<K>::mul_clear(ClearInteger<K>)      -> SecretInteger<K>;
```

8.13 ClearFixed

A clear fixed point number is represented as a `ClearInteger<K>` value x and a constant F, with the fixed point number so represented being $x/2^F$. This representation, and the associated functions, should be avoided if at all possible. The `ClearIEEE` is much faster, providing near native speed operations. The `ClearFixed` type is really for interacting with the `SecretFixed` type. We provide a full math library for this type, but this again is mainly for interaction/testing purposes.

```
use scale_std::fixed_point::*;
#[inline(always)]
#[scale::main(KAPPA = 40)]
fn main() {
  let c: ClearFixed<40,20>= ClearFixed::from(3.141592);
  print!(c,"\n");
}
```

`ClearFixed::from(x)`

You can convert an `f64` constant, an **i64**, a **ClearModp**, a `ClearIEEE` or a `ClearInteger<K>` to a `ClearFixed<K,F>` as follows:

```
let two = ClearModp::from(2);
let two_as_cfix: ClearFixed<40,20>= ClearFixed::from(c);
```

`a.recast()`

A value of one size can be recast to another using the `recast` member function. It is assumed the user knows what they are doing here and that such recasting will necessarily result in a valid representation being created.

```
let c = ClearModp::from(2);
let ci: ClearFixed<40,20>= ClearFixed::from(c);
let ci1: ClearFixed<50,30>= unsafe{ a.recast() };
```

`a.rep()`

Returns the internal representation as a `ClearInteger<K>` value, this is basically the fixed point value multiplied by 2^F.

`a.rep_integer()`

This returns the `ClearModp` value which is the integer part of the fixed point number. Basically it executes $x >> F$ if x is positive and $-((-x) >> F)$ if x is negative.

`a.set(v)`

Sets the internal representation to be the `ClearInteger<K>` value v.

`a.set_modp(v)`

Sets the internal representation to be the `ClearInteger<K>` value obtianed by calling `set` for the type `ClearInteger<K>` on the `ClearModp` value v.

ClearFixed::randomize()

Produces a random value in the range $[0, \ldots, 1)$, it assumes that $K > F$. This random number is the 'same' for all players.

```
let a: ClearFixed<20,10> = ClearFixed::randomize();
```

8.13.1 Comparisons

The following 'operators' can be applied between two `ClearFixed<K,F>` values,the output being a **ClearModp** value. As the result of the operator cannot be used in a conditional branch, we use the member function notation for such 'operators', as opposed to the operator notation. Thus syntactically the programmer is less likely to make a mistake.

a.gt(x), a.ge(x), a.lt(x), a.le(x), a.gt(x), a.eq(x), a.ne(x)

The following give variants which compare just to zero.

a.gtz(), a.gez(), a.ltz(), a.lez(), a.gtz(), a.eqz(), a.nez(x)

8.14 SecretFixed

A secret fixed point number is represented as a `SecretInteger<K>` value x and a constant F, with the fixed point number so represented being $x/2^F$. Many operations on `SecretFixed` values make use of the statistical security gap `KAPPA` defined in the pre-amble.

```
use scale_std::fixed_point::*;
#[inline(always)]
#[scale::main(KAPPA = 40)]
fn main() {
  let c: SecretFixed<40,20>= SecretFixed::from(3.141592);
  print!(c,"\n");
}
```

SecretFixed::from(x)

You can convert an `f64` constant, an **i64**, **ClearModp**, `ClearInteger<K>` or a `ClearFixed<K,F>` to a `SecretFixed<K,F>` as follows:

```
let two = ClearModp::from(2);
let two_as_cfix: ClearFixed<40,20>= ClearFixed::from(c);
let two_as_sfix: SecretFixed<40,20>= SecretFixed::from(c);
```

a.recast()

A value of one size can be recast to another using the `recast` member function. It is assumed the user knows what they are doing here and that such recasting will necessarily result in a valid representation being created.

```
let c1: SecretFixed<40,20>= ClearFixed::from(c);
let c2: SecretFixed<50,30>= unsafe{ a.recast() };
```

a.rep()

Returns the internal representation as a SecretInteger<K> value, this is basically the fixed point value multiplied by 2^F.

a.set(v)

Sets the internal representation to be the SecretInteger<K> value v.

a.set_modp(v)

Sets the internal representation to be the SecretInteger<K> value obtianed by calling set for the type SecretInteger<K> on the SecretModp value v.

SecretFixed::randomize()

Produces a (secret) random value in the range $[0, \ldots, 1)$, it assumes that $K > F$. This random number is the 'same' for all players.

```
let a: SecretFixed<20,10> = SecretFixed::randomize();
```

8.14.1 Comparisons

The following 'operators' can be applied between two SecretFixed<K,F> values,the output being a **SecretModp** value. As the result of the operator cannot be used in a conditional branch, we use the member function notation for such 'operators', as opposed to the operator notation. Thus syntactically the programmer is less likely to make a mistake.

a.gt(x), a.ge(x), a.lt(x), a.le(x), a.gt(x), a.eq(x), a.ne(x)

The following give variants which compare just to zero.

a.gtz(), a.gez(), a.ltz(), a.lez(), a.gtz(), a.eqz(), a.nez(x)

8.15 Arithmetic of ClearFixed and SecretFixed

Standard arithmetic operations $+, -, \times, /$ can be applied to types ClearFixed<K,F> and SecretFixed<K,F>. To ensure efficiency *no check is performed for overflow*. Thus if the number exceed 2^{K-F-1} in absolute value then undefined behvaiour is likely to result. Clear and secret values can be combined to result in a secret value.

8.16 ClearFloat

A clear floating point number is represented by a an Array of five ClearModp-elements: (v, p, z, s, err). They represent the value

$$u = (1 - 2 \cdot s) \cdot (1 - z) \cdot v \cdot 2^p,$$

with the convention that if $z = 1$ then $s = 0$. Note, that as Arrays do not implement Copy neither does ClearFloat, it does however implement Clone. The ClearFloat type is meant for interacting with the SecretFloat type. It is very slow in comparison to the ClearIEEE and ClearFixed types.

The type comes with two parameters V and P, which describe the bitsize of the mantissa and the exponent respectively. An optional parameter is DETECT_OVERFLOW (default value: true), that indicates whether mathematical operators should check if their output exceeds the bounds of V and P. To set the DETECT_OVERFLOW value to false you should specify this in the macro before the main call, as in....

```
use scale_std::floating_point::*;
#[inline(always)]
#[scale::main(KAPPA = 40, DETECT_OVERFLOW = false)]
fn main() {
  let c: ClearFloat<40,10>= ClearFloat::from(3.141592);
  print!(c,"\n");
}
```

If the value is not defined in the initial macro then the default value of true is taken:

```
use scale_std::floating_point::*;
#[inline(always)]
#[scale::main(KAPPA = 40)]
fn main() {
  let c: ClearFloat<40,10>= ClearFloat::from(3.141592);
  print!(c,"\n");
}
```

To avoid unnecessary cloning, the `Print` trait also allows references to a `ClearFloat` to be passed to it:

```
use scale_std::floating_point::*;
#[inline(always)]
#[scale::main(KAPPA = 40)]
fn main() {
  let c: ClearFloat<40,10>= ClearFloat::from(3.141592);
  print!(&c,"\n");
}
```

`ClearFloat::from(x)`

You can convert a value of type `f64`, **i64**, `ClearFixed<K,F>`, `ClearInteger<K>`[2] or `ClearIEEE` to `ClearFloat`.

```
let two_as_cfloat: ClearFixed<40,10>= ClearFloat::from(2_f64);
```

Note that the `K` for `ClearInteger<K>` and the `K` for `ClearFixed<K,F>` must be larger than the `V` of `ClearFloat<V,P>`.

`ClearFloat::set(x)`

Given an `Array` of five **ClearModp** values this sets them to be the representation of a `ClearFloat` value.

`a.v(), a.p(), a.z(), a.s(), a.err()`

Gets the associated representation value as a **ClearModp**.

8.16.1 Comparisons

The following 'operators' can be applied between two `ClearFloat<V,P>` values,the output being a **ClearModp** value. As the result of the operator cannot be used in a conditional branch, we use the member function notation for such 'operators', as opposed to the operator notation. Thus syntactically the programmer is less likely to make a mistake.

`a.gt(x), a.ge(x), a.lt(x), a.le(x), a.gt(x), a.eq(x), a.ne(x)`

The following give variants which compare just to zero.

[2] Note here K does not necessarily have to equal V.

a.gtz(), a.gez(), a.ltz(), a.lez(), a.gtz(), a.eqz(), a.nez(x)

To be able to avoid costly cloning, the comparisons can be used on both references and clones. All options for `gt()` and `gtz()` for `ClearModp` are presented in following example:

```
let ca: ClearFloat<40,20> = ClearFloat::from(123546);
let cb: ClearFloat<40,20> = ClearFloat::from(789);

let gtcc = ca.clone().gt(cb.clone());
let gtrc = (&ca).gt(cb.clone());
let gtcr = ca.clone().gt(&cb);
let gtrr = (&ca).gt(&cb);

let gtzc = ca.clone().gtz();
let gtzr = (&ca).gtz();
```

8.17 SecretFloat

A secret floating point number is represented by an Array of five `SecretModp`-values. Together they represent the value as follows:

$$u = (1 - 2 \cdot s) \cdot (1 - z) \cdot v \cdot 2^p,$$

again with the convention that if $z = 1$ then $s = 0$. In this array, the mantissa v has a bitsize `V` and the exponent p is of size `P`. Many operations on `SecretFloat` values make use of the statistical security gap `KAPPA` defined in the preamble. As for the `ClearFloat` type, there is also a parameter `DETECT_OVERFLOW` defined in the same manner. Note, that as Arrays do not implement `Copy` neither does `SecretFloat`, it does however implement `Clone`.

```
use scale_std::floating_point::*;
#[inline(always)]
#[scale::main(KAPPA = 40)]
fn main() {
  let c: SecretFloat<40,20>= SecretFloat::from(3.141592);
  print!(c.reveal(),"\n");
}
```

SecretFloat::from(x)

You can convert a value of type `f64`, **i64**, `SecretInteger<K>` or `SecretFixed<K,F>` to `SecretFloat` as follows:

```
let two_as_sfloat: SecretFloat<40,10> = SecretFloat::from(2_f64);
```

Note that the `K` for `SecretInteger<K>` and the `K` for `SecretFixed<K,F>` must be larger than the `V` of `SecretFloat<V,P>`.

SecretFloat::set(x)

Given an `Array` of five **SecretModp** values this sets them to be the representation of a `SecretFloat` value.

a.v(), a.p(), a.z(), a.s(), a.err()

Gets the associated representation value as a **SecretModp**.

a.reveal()

You can reveal a value of type `SecretFloat` to a `ClearFloat`-type value as follows:

```
let cfa: ClearFloat<40,10> = sfa.reveal();
```

If the entered value is invalid (err == 0), `reveal()` returns a `ClearFloat` with all parameter values set to zero. This to avoid information leakage.

8.17.1 Comparisons

The following 'operators' can be applied between two `SecretFloat<V,P>` values,the output being a **SecretModp** value. As the result of the operator cannot be used in a conditional branch, we use the member function notation for such 'operators', as opposed to the operator notation. Thus syntactically the programmer is less likely to make a mistake.

a.gt(x), a.ge(x), a.lt(x), a.le(x), a.gt(x), a.eq(x), a.ne(x)

The following give variants which compare just to zero.

a.gtz(), a.gez(), a.ltz(), a.lez(), a.gtz(), a.eqz(), a.nez(x)

To be able to avoid costly cloning, the comparisons can be used on both references and clones. All options for `gt()` and `gtz()` for `SecretModp` are presented in following example:

```
let sa: SecretFloat<40,20> = SecretFloat::from(123546);
let sb: SecretFloat<40,20> = SecretFloat::from(789);

let gtcc = sa.clone().gt(sb.clone());
let gtrc = (&sa).gt(sb.clone());
let gtcr = sa.clone().gt(&sb);
let gtrr = (&sa).gt(&sb);

let gtzc = sa.clone().gtz();
let gtzr = (&sa).gtz();
```

8.18 Arithmetic of ClearFixed and SecretFixed

Standard arithmetic operations $+, -, \times, /$ can be applied to types `ClearFloat<V,P>` and `SecretFloat<V,P>`. Depending on the flag parameter `DETECT_OVERFLOW`, the results of these operations is checked for overflow/underflow. Clear and secret values can be combined to result in a secret value.

To avoid the costly `.clone()` operation on every input to arithmetic operations (and comparisons), the operators also accept references. This allows the variable to be used in arithmetic without ending its lifetime. These are accessed as in the following example for addition.

```
let ca: ClearFloat<40,20>  = ClearFloat::from(123456);
let sa: SecretFloat<40,20> = SecretFloat::from(ca.clone());
let cb: ClearFloat<40,20>  = ClearFloat::from(789);
let sb: SecretFloat<40,20> = SecretFloat::from(cb.clone());

let sscr = sa.clone() + &sb;
let ssrr = &sa + &sb;
let ssrc = &sa + sb.clone();
```

```
let sscc = sa.clone() + sb.clone();

let sccr = sa.clone() + &cb;
let scrr = &sa + &cb;
let scrc = &sa + cb.clone();
let sccc = sa.clone() + cb.clone();

let cscr = ca.clone() + &sb;
let csrr = &ca + &sb;
let csrc = &ca + sb.clone();
let cscc = ca.clone() + sb.clone();

let cccr = ca.clone() + &cb;
let ccrr = &ca + &cb;
let ccrc = &ca + cb.clone();
let cccc = ca.clone() + cb.clone();
```

8.19 ORAM Operations

Given a `SecretModp` index value i and an `Array` or `Slice` A one wants to either read the value $A[i]$, or one wants to update $A[i]$ with a new value v. One would want to do this without revealing the value i, or even any access pattern. This is enabled via a Demux data structure which provided a linear time read and write access to the data structure in an oblivious way. The key thing needed is an upper bound on the bit length of the index value i, which is kind of known in any case, as i must be less than the length of the `Array` or `Slice`.

Note these operations require a bit length of the base prime of larger than 100.

a.read_oram::<K,KAPPA>(i)

This takes a secret index of bit length less than K are does a read into the array a at position i. The value K must be such that 2^K is greater than or equal to the length of a. As an example, we present an example for `Array`s, an identical syntax works for `Slice`s,

```
let mut arr : Array<SecretModp, 20> = Array::uninitialized();
for i in 0..20 {
    arr.set(i, &SecretModp::from(i as i64));
}

let i = SecretModp::from(7);
let v = arr.read_oram::<5,40>(i);
```

a.write_oram::<K,KAPPA>(i, v)

This takes a secret index of bit length less than K are does a write into the array a of the value v at position i. The value K must be such that 2^K is greater than or equal to the length of a. As an example, we present an example for `Slice`s, an identical syntax works for `Array`s,

```
let mut sli : Slice<SecretModp> = Slice::uninitialized(20);
for i in 0..20 {
    sli.set(i, &SecretModp::from(i as i64));
}

let i = SecretModp::from(13);
let v = SecretModp::from(666);
```

```
sli.write_oram::<5,40>(i, v);
```

Demux::<K,KAPPA>(i)

In order to allow efficient indexing at the same index, we also allow the creation of the demux data externally. This function produces a `Slice` of `SecretModp` values of length at least 2^K, which corresponds to the demux array of the index i. Note, the function outputs an `Slice` which is bigger than needed. This is done in order to enable a memory efficient algorithm.

demux::<KAPPA>(i, k)

Same, but now k is a runtime variable and not a compile time constant.

a.read_with_demux(&demux)

This is the same as the `read_oram` operation, but uses the demux output. Thus we can execute

```
let i = SecretModp::from(19);
let demx = Demux::<5,40>(i);
let va = arr.read_with_demux(&demx);
let vs = sli.read_with_demux(&demx);
```

a.write_with_demux(&demux, v)

This is the same as the `write_oram` operation, but uses the demux output. Thus we can execute

```
let i = SecretModp::from(19);
let demx = Demux::<5,40>(i);
let v = SecretModp::from(666);
let va = arr.write_with_demux(&demx, v);
let vs = sli.write_with_demux(&demx, v);
```

a.read_with_demux_and_offset(offset, &demux)

This is the same as the `read_oram_with_demux` operation, but offsets the demux operation to start at position `offset`. No checking is performed as to whether this gives array out of bounds errors.

```
let i = SecretModp::from(5);
let demx = demux::<40>(i, 3);
let va = arr.read_with_demux_and_offset(13, &demx);
let vs = sli.read_with_demux_and_offset(13, &demx);
```

a.write_with_demux_and_offset(offset, &demux, v)

This is the same as the `write_oram_with_demux` operation, but offsets the demux operation to start at position `offset`. No checking is performed as to whether this gives array out of bounds errors.

```
let i = SecretModp::from(5);
let demx = demux::(i,3);
let v = SecretModp::from(666);
let va = arr.write_with_demux(13, &demx, v);
let vs = sli.write_with_demux(13, &demx, v);
```

9 Math Library

Eventually we expect every floating point type to have a full set of math functions. Remember the `ClearIEEE` type is must faster than the `ClearFixed<K,F>` type, but the `SecretFixed<K,F>` is generally much faster than the `SecretIEEE` type. To include the functions in the math library use

```
use scale_std::math::*;
```

This captures traits `Floor`, `FAbs` and `Sqrt` (implementing member functions `floor()`, `fabs()` and `sqrt()` respectively), plus also a supertrait called `Float` which captures much of the properties common to all floating point types.

Currently the following are available (we comment the function out if it is not yet fully tested/implemented):

9.1 Generic Routines

`poly_eval(poly, x)`

This generic function with signature

```
pub fn poly_eval<S, C, const N: u64>(poly: Array<C, N>, x: S) -> S
where
  S: Float,
  S: Add<C, Output = S>,
  S: Mul<S, Output = S>,
  C: Float,
```

allows one to evaluate a polynomial in type `C` (a clear floating point type) at a value of type `S` (which is either the *same* clear floating point type, or its *associated* secret variant).

`Pade(P, Q, x)`

This generic function with signature

```
pub fn Pade<S, C, const N: u64>
 (poly_p: Array<C, N>, poly_q: Array<C, N>, x: S) -> S
where
  S: Float,
  S: Add<S, Output = S>,
  S: Mul<S, Output = S>,
  S: Div<S, Output = S>,
  S: Mul<C, Output = S>,
  S: From<C>,
  C: Float,
```

allows one to evaluate the Pade approximation given by polynomials P and Q in type `C` (a clear floating point type) at a value of type `S` (which is either the *same* clear floating point type, or its *associated* secret variant). The polynomials P and Q must have the same degree.

All floating point types `T` have predefined constants

```
T::two_pi();
T::pi();
T::half_pi();
T::ln2();
T::e();
```

9.2 ClearIEEE

The following are all implemented via a local function call to the SCALE C++ runtime; thus are relatively fast.

```
let c = ClearIEEE::from(0.5431);
let t = c.acos();
let t = c.asin();
let t = c.atan();
let t = c.cos();
let t = c.cosh();
let t = c.sin();
let t = c.sinh();
let t = c.tan();
let t = c.tanh();
let t = c.exp();
let t = c.exp2()
let t = c.log();
let t = c.log2();
let t = c.log10();
let t = c.ceil();
let t = c.fabs();
let t = c.floor();
let t = c.sqrt();
```

Note, `exp2()` computes the function 2^x.

9.3 SecretIEEE

The following functions are implemented

```
let s = SecretIEEE::from(0.5431);
let t = s.acos();
let t = s.asin();
let t = s.atan();
let t = s.cos();
let t = s.cosh();
let t = s.sin();
let t = s.sinh();
let t = s.tan();
let t = s.tanh();
let t = s.exp();
let t = s.exp2()
let t = s.log();
let t = s.log2();
let t = s.log10();
let t = s.ceil();
let t = s.fabs();
let t = s.floor();
let t = s.sqrt();
```

Note logarithms of negative values result in undefined behaviour.

9.4 ClearFixed

The following functions are implemented

```
let c: ClearFixed<40,20>= ClearFixed::from(0.5431);
let t = c.acos();
let t = c.asin();
let t = c.atan();
let t = c.cos();
let t = c.cosh();
let t = c.sin();
let t = c.sinh();
let t = c.tan();
let t = c.tanh();
let t = c.exp();
let t = c.exp2()
let t = c.log();
let t = c.log2();
let t = c.log10();
let t = c.ceil();
let t = c.fabs();
let t = c.floor();
let t = c.sqrt();
```

9.5 SecretFixed

The following functions are implemented

```
let s: SecretFixed<40,20>= SecretFixed::from(0.5431);
let s = c.acos();
let s = c.asin();
let s = c.atan();
let s = c.cos();
let s = c.cosh();
let s = c.sin();
let s = c.sinh();
let s = c.tan();
let s = c.tanh();
let s = c.exp();
let s = c.exp2()
let s = c.log();
let s = c.log2();
let s = c.log10();
let s = c.ceil();
let s = c.fabs();
let s = c.floor();
let s = c.sqrt();
```

9.6 ClearFloat

The following functions are implemented. For the exponential and (hence) the hyperbolic functions we assume that $V < 2^{P-1}$, which it will be in almost all instances; unless P is very small.

```
let c: ClearFloat<40,20>= ClearFloat::from(0.5431);
// let t = c.clone().acos();
// let t = c.clone().asin();
```

```
// let t = c.clone().atan();
// let t = c.clone().cos();
let t = c.clone().cosh();
// let t = c.clone().sin();
let t = c.clone().sinh();
// let t = c.clone().tan();
let t = c.clone().tanh();
let t = c.clone().exp();
let t = c.clone().exp2()
let t = c.clone().log();
let t = c.clone().log2();
let t = c.clone().log10();
let t = c.clone().ceil();
let t = c.clone().fabs();
let t = c.clone().floor();
let t = c.clone().sqrt();
```

9.7 SecretFloat

The following functions are implemented. For the exponential and (hence) the hyperbolic functions we assume that $V < 2^{P-1}$, which it will be in almost all instances; unless P is very small.

```
let s: SecretFloat<40,20>= SecretFloat::from(0.5431);
// let t = s.clone().acos();
// let t = s.clone().asin();
// let t = s.clone().atan();
// let t = s.clone().cos();
let t = s.clone().cosh();
// let t = s.clone().sin();
let t = s.clone().sinh();
// let t = s.clone().tan();
let t = s.clone().tanh();
let t = s.clone().exp();
let t = s.clone().exp2()
let t = s.clone().log();
let t = s.clone().log2();
let t = s.clone().log10();
let t = s.clone().ceil();
let t = s.clone().fabs();
let t = s.clone().floor();
let t = s.clone().sqrt();
```