

# Modern Approaches to Subrange Types, Memory Management, and Arbitrary-Precision Arithmetic

Neel Joshi

McMaster University  
joshin10@mcmaster.ca

**Abstract.** This survey reviews how modern languages and runtimes address three recurring problems: expressing simple value constraints in types, managing memory with predictable costs, and supporting integers that grow beyond machine word sizes. It compares subrange and refinement mechanisms in Haskell (Liquid Haskell), Java (Checker Framework), C++ (Contracts), and Python (type hints). It then surveys memory techniques that aim for predictable behavior, including arena allocation, escape analysis for stack allocation, and the WebAssembly execution model. Finally, it outlines how Python and Java represent big integers internally and when they switch between multiplication algorithms such as Karatsuba, Toom-Cook, and Schönhage–Strassen. The goal is to summarize the main trade-offs of architectures between language implementations.

## 1 Introduction

Many languages and runtimes have tried to balance two goals: making software easier to build and making its behavior easier to predict. Managed runtimes reduce many memory errors by using garbage collection (GC), and most systems still rely on fixed-width machine integers for performance. At the same time, modern applications in security, data processing, and low-latency systems often benefit from stronger guarantees (for example, “this index is in range”), more explicit control over allocation costs, and integers that do not overflow.

This survey focuses on three themes that show up across many ecosystems:

**Value Constraints in Types.** How subrange and refinement types express bounds and related properties.

**Predictable Allocation and Cleanup.** How systems manage intermediate values and memory without long, unpredictable pauses.

**Big Integers.** How arbitrary-precision arithmetic is represented and implemented efficiently.

This survey first reviews subrange and refinement types (Section 2). It then discusses where intermediate values live during execution (Section 3) and approaches to memory management that avoid full tracing GC (Section 4). It closes with a survey of arbitrary-precision arithmetic and multiplication algorithms (Section 5).

## 2 Subrange Types and Refinement Types

This section explores how modern tools use types to enforce simple rules on data, such as ensuring a number stays between 0 and 255 or that an array index is valid. Subrange types handle basic limits, while refinement types go further by attaching specific conditions to a standard type. Instead of checking these rules while the program runs, many tools translate the program state into mathematical constraints (SMT formulas) and use automated theorem provers to verify that the code satisfies its specifications for all possible inputs.

### 2.1 Theoretical Foundations: From Subranges to Refinements

A subrange type is formally defined as a type  $T'$  derived from a base type  $T$  such that every value  $v \in T'$  satisfies a constraint  $P(v)$ , where  $P$  is a predicate. In the simplest case,  $P$  checks bounds:  $L \leq v \leq U$ . Historically, languages like Pascal offered subrange types, but these were often checked at runtime or limited to constant bounds [1].

Modern systems employ *refinement types*, which generalize subranges. A refinement type is typically denoted as  $\{v : B \mid \phi(v)\}$ , where  $B$  is a basic type (like `Int` or `Bool`) and  $\phi$  is a logical formula constraining the value  $v$  [2]. Unlike full dependent types, most refinement-type tools restrict  $\phi$  to solver-friendly fragments (for example, linear arithmetic), so the checks can run automatically during compilation.

The verification process generally involves generating *verification conditions* (VCs): small logical statements that must be true for the program to be safe. If a function  $f$  requires an argument of type  $\{v : \text{Int} \mid v > 0\}$  and is called with a value  $x$ , the checker generates the logical implication “Context  $\implies x > 0$ .” If the solver can prove this implication, the code type-checks; otherwise, the tool reports an error [3].

### 2.2 Haskell: Liquid Haskell and SMT Integration

Haskell already has strong static typing. *Liquid Haskell* adds refinement types by checking extra annotations during compilation (as a plugin to GHC). This allows programmers to state and check invariants that the standard Haskell type system cannot express [4].

**Implementation Mechanics.** Liquid Haskell works by parsing special comments in the source code. It extracts refinement specifications and generates VCs from Haskell Core (the compiler’s intermediate representation). For example, defining natural numbers:

$\{-\text{type Nat} = \{v : \text{Int} \mid v \geq 0\} \text{-}\}$

Contracts (pre- and post-conditions) are expressed as refined function types. If a function `div` requires a non-zero divisor:

```
{-@ div :: n:Nat -> d:{v:Nat | v != 0} -> Int @-}
```

When `div` is called, Liquid Haskell checks that the call site provides a value that meets the function’s input constraints. Internally, it reduces this check to a logical implication that an external solver (for example, Z3) attempts to prove [5].

**Worked Example: Safe List Operations.** Plain Haskell exposes many useful but partial functions. For example, `head :: [a] -> a` and `(!!) :: [a] -> Int -> a` can fail at runtime on empty lists or out-of-range indices. Liquid Haskell can encode these safety conditions in types and check them statically.

A common technique is to introduce a *measure*: a pure function that Liquid Haskell can use inside refinements. The measure below models list length. Using it, `safeHead` requires a non-empty list, and `index` requires an index strictly smaller than the list length:

```
{-@ measure len @-}
len :: [a] -> Int
len [] = 0
len (_,xs) = 1 + len xs

{-@ type Nat = {v:Int | v >= 0} @-}

{-@ safeHead :: {xs:[a] | len xs > 0} -> a @-}
safeHead :: [a] -> a
safeHead (x:_) = x

{-@ index :: xs:[a] -> {i:Nat | i < len xs} -> a @-}
index :: [a] -> Int -> a
index (x:_) 0 = x
index (_,xs) i = index xs (i - 1)
```

If a program tries to call `safeHead []` or `index [1,2,3] 10`, Liquid Haskell reports a type error during checking, even though the code is valid Haskell syntax. This is an example of an invariant—“the list is non-empty” or “the index is in bounds”—that is not captured by standard Haskell types.

**Refinement Reflection.** A notable feature in Liquid Haskell is *refinement reflection*, which allows some Haskell functions to be used in the refinement logic. If a function `fib` is defined in Haskell, it can be “reflected” into the logic, allowing specifications like  $\{v : \text{Int} \mid v = \text{fib } n\}$  [6]. In practice, this can help express and check properties of data structures (for example, that a Map lookup is only performed when a key is present), as long as the reflected definitions remain within what the tool can reason about.

### 2.3 Java: The Checker Framework and Pluggable Types

Java’s built-in type checking centers on classes and interfaces, but since Java 8 the language also supports type annotations (JSR 308), which can appear on

many uses of a type (e.g., `List<@NonNull String>`). The Checker Framework leverages these annotations to implement opt-in, pluggable type systems [7].

**Architecture of Pluggable Types.** The Checker Framework does not change the JVM or the Java language syntax; it operates as a compiler plugin (an annotation processor). It uses dataflow analysis to track facts about values along different control-flow paths.

**Qualifiers.** Annotations like `@IntRange(from=0, to=255)` define the sub-range.

**Type Factory.** Determines the type of expressions based on their constituent parts. For example, if `x` is `@IntRange(0, 10)` and `y` is `@IntRange(0, 10)`, the expression `x + y` is inferred to have type `@IntRange(0, 20)`.

**Visitor.** The framework traverses the program’s syntax tree (often called an abstract syntax tree, or AST) to enforce subtype relationships. If a variable of type `@IntRange(0, 100)` is assigned to a parameter requiring `@IntRange(0, 50)`, the framework flags an error [7].

**Practical Application: The Index Checker.** One of the standard checkers is the Index Checker, which enforces that integer variables used as array indices are non-negative and less than the array’s length.

```
void process(int[] arr, @IndexFor("arr") int i) {
    System.out.println(arr[i]); // Safe by construction
}
```

The framework uses a specialized type hierarchy where `@IndexFor("arr")` is a subtype of `@NonNegative`. The dataflow analysis handles control flow, refining types inside `if` blocks. For instance, inside `if (i < arr.length)`, `i` is refined to satisfy the upper bound constraint [8]. This kind of check can prevent `ArrayIndexOutOfBoundsException` and helps catch out-of-bounds bugs before running the program.

## 2.4 C++: Template Metaprogramming and Contracts (C++26)

C++ approaches subrange guarantees in two common ways: encoding bounds in templates so the compiler can reason about them, and stating explicit preconditions and postconditions through the upcoming Contracts facility.

**Bounded Integers via Templates.** Libraries like `bounded::integer` use C++’s template system to encode ranges into the type signature itself: `bounded::integer<Min, Max>`.

**Arithmetic Operations.** When two bounded integers are added, the result is a new type with bounds expanded to cover the worst-case scenario. `bounded::integer<0, 10> + bounded::integer<1, 5>` results in `bounded::integer<1, 15>` [9].

**Overflow Safety.** Because the bounds are tracked at compile-time, the compiler can prevent operations that would overflow the underlying machine representation (e.g., `int`) or force the user to handle the overflow explicitly. The compiler can also choose the smallest storage type (e.g., `uint8_t`) that fits the range, optimizing memory usage [10].

**Metaprogramming Limitations.** While powerful, this approach can increase compilation time and produce verbose error messages, because the compiler must expand and evaluate many templates before it can type-check the final program.

**Contracts in C++26.** The C++26 standard is poised to introduce *Contracts*, a language feature for specifying preconditions (`pre`), postconditions (`post`), and invariants (`contract_assert`) [11].

**Syntax.**

```
int divide(int n, int d)
  pre (d != 0)
  post (r: r * d == n);
```

**Semantics.** Unlike a simple `assert`, a contract is part of the function interface. Depending on build settings, a contract can be enforced (terminate on failure), observed (log but continue), or ignored (allowing the compiler to optimize assuming the condition is true). This allows subrange constraints to be documented and checked without relying on external tools or complex template syntax [12,13].

## 2.5 Python: Gradual Typing and Runtime Validation

Python remains dynamically typed at runtime, but since PEP 484 it supports optional type hints. While the Python runtime does not enforce these hints, static analyzers (like MyPy) and runtime validators (like Pydantic) can use them.

**Annotated Types.** Introduced in PEP 593, **Annotated** allows metadata to be attached to types [14]. Static analysis tools can consume this to flag violations.

**Runtime Validation.** Libraries like Pydantic parse these annotations to generate runtime validation logic. When data enters the system (e.g., via a REST API), Pydantic can check that it matches the declared constraints. This does not provide compile-time guarantees, but it is a practical safety check at system boundaries [15].

## 2.6 WebAssembly: Interface Types and Validation

WebAssembly (Wasm) enforces type safety at a lower level. Its validation algorithm ensures that the operand stack remains consistent in terms of types and depths.

**Numeric Types.** Wasm supports `i32`, `i64`, `f32`, `f64`. While it lacks native subrange types in the core instruction set, the Interface Types proposal (and the

Component Model) introduces higher-level types like `u8`, `s16`, and variants for interoperability [16].

**Validation.** When a Wasm module is loaded, the engine validates that every instruction sequence is type-safe. For example, an `i32.add` instruction must always be preceded by pushing two `i32` values onto the stack. This single-pass validation is crucial for the security of the web platform, ensuring that Wasm code cannot access memory or stack slots illegally [17].

### 3 Memory Allocation for Intermediate Results

This section surveys how runtimes and compilers handle intermediate results—values that exist briefly between operations. Where these values live (registers, stack slots, or heap objects) strongly affects performance, so modern systems try to keep them cheap to create and access.

#### 3.1 The WebAssembly Stack Machine Model

WebAssembly is defined as a virtual stack machine. This design choice was made to ensure compact binary encoding and easy validation, but it requires careful mapping to physical registers by the execution engine.

**Operand Stack vs. Call Stack.** Wasm distinguishes between the operand stack (used for expression evaluation) and the call stack (used for function frames).

**Instructions.** Operations like `i32.add` pop two values from the operand stack and push the result.

**Implicit Registers.** In a high-performance Wasm engine like V8 (TurboFan) or SpiderMonkey, the operand stack does not physically exist in memory during execution. Instead, an abstract interpretation pass during compilation maps these stack slots to physical CPU registers. If the stack depth exceeds the number of available registers, values are “spilled” to the physical stack [18].

**Determinism.** The depth of the operand stack is statically determined at every point in the code. This prevents stack underflows/overflows at runtime and allows the just-in-time (JIT) compiler to allocate stack frames of a fixed size [19].

**Latency Hiding in Stack Machines.** While stack machines can create long dependency chains, real engines compile Wasm to a lower-level representation. They often translate the implicit stack into named values in static single assignment (SSA) form (each value is assigned once) inside the compiler’s intermediate representation (IR). Once the values are explicit, the compiler can reorder independent instructions to overlap slow operations (like memory loads) with other work, a common technique called *latency hiding* [20].

### 3.2 Register Allocation and Spilling

When there are more live values than CPU registers, the compiler must choose which values stay in registers and which are temporarily written to memory (“spilled”). This process is called *register allocation*.

**Evolution of Allocators. Linear Scan.** Popular in JIT compilers (like early Java HotSpot) for its speed. It allocates registers in a single pass over the code. It is fast but often produces suboptimal code with excessive spilling.

**Graph Coloring.** Models allocation as coloring an interference graph (where nodes are variables and edges represent overlapping lifetimes). It often produces high-quality code, but it is computationally expensive; the general problem is *NP*-complete, so compilers use heuristics [21].

**Greedy Allocator (LLVM).** LLVM transitioned to a greedy register allocator. It assigns registers to live ranges based on “spill weight” (calculated from access frequency, e.g., variables in loops have higher weight). It employs *live range splitting*: if a variable is live across a large range but active only in short bursts, the allocator splits the live range. It might reside in a register during a loop and be spilled to the stack during inactive periods [22].

**Spilling and Security.** Spilling involves moving a value from a register to RAM (stack). This has significant performance costs (cache latency) and security implications. Recent research has highlighted that register spilling can introduce side-channel vulnerabilities. If a sensitive intermediate result (e.g., an AES round key) is spilled to the stack, it may be recoverable from memory or cache even after the register is cleared. Secure register allocation attempts to minimize spilling of sensitive variables or ensure they are overwritten immediately after use [23].

### 3.3 C++: Return Value Optimization (RVO) and Copy Elision

In C++, intermediate results of object types can incur significant overhead if copied. *Return Value Optimization* (RVO) is a compiler technique that eliminates these copies.

**Mechanism.** When a function returns an object by value, the compiler essentially rewrites the function to accept a hidden pointer to the memory location where the result should be stored (in the caller’s stack frame). The function constructs the object directly in that location [24].

**Guaranteed Elision.** Since C++17, the standard requires copy elision in many common cases involving temporaries. This means `T x = T(T(f()))`; can involve zero copies, regardless of the complexity of `T` [25].

This allows C++ developers to return large objects (like matrices or vectors) from functions while still getting efficient code, because the result can be constructed directly in the caller’s storage.

### 3.4 Escape Analysis: Stack Allocation in Managed Runtimes

Even in a garbage-collected runtime, the compiler can avoid some heap allocations. In Java, the JIT compiler can use *escape analysis* to decide when an object can be stack-allocated (or removed entirely).

**Analysis.** The JIT compiler (HotSpot C2) analyzes the scope of a new object. If the object’s reference does not “escape” the method (e.g., it is not returned or assigned to a global field), the compiler can optimize the allocation [26].

**Scalar Replacement.** The compiler may go further and break the object into its constituent fields (scalars), keeping them in registers or stack slots. This effectively eliminates the object allocation entirely. The object header is never created, and no heap allocation occurs [27].

**Limits.** Escape analysis happens at runtime during JIT compilation. If the method is too complex or the object escapes in a rare branch (e.g., an exception path), the optimization fails, falling back to the heap. This unpredictability is why explicit arenas are preferred in systems programming [28].

## 4 Memory Management Without Full Garbage Collection

Garbage collection (GC) simplifies memory management, but it can introduce pauses when the runtime traces and frees unreachable objects. When predictable response time matters, developers often use techniques that make allocation and freeing more explicit. This section surveys a few approaches that reduce or avoid full tracing GC.

### 4.1 Region-Based Memory Management (Arenas)

Region-based memory management (or Arena allocation) groups objects with similar lifetimes into a contiguous block of memory.

**Mechanism and Algorithms.** An Arena is a large pre-allocated block of memory.

**Allocation.** A simple pointer bump. The allocator maintains a pointer to the beginning of the free space. Allocating an object is simply incrementing this pointer (`ptr += size`). This is an  $O(1)$  operation, comparable to stack allocation and significantly faster than `malloc/new`, which must search free lists [29].

**Deallocation.** Individual objects are never freed. Instead, the entire region is reset or freed at once when a logical unit of work (e.g., an HTTP request, a video frame) completes.

**Cache Locality.** Because objects are allocated sequentially, they are likely to be contiguous in memory, significantly improving CPU cache hit rates and reducing address-translation misses [30].



**Implementation in C++ and Protocol Buffers.** Google’s Protocol Buffers library for C++ uses arenas extensively. Parsing a message allocates all sub-objects (strings, nested messages) in a single arena. When the message is processed, the arena is reset. This avoids the overhead of thousands of small `new/delete` calls and memory fragmentation.

The C++17 standard introduced `<memory_resource>` and `std::pmr` (Polymorphic Memory Resources), which standardizes this pattern. Developers can use `std::pmr::monotonic_buffer_resource` to back standard containers like `std::vector` with an arena. This allows a `std::vector` to allocate its internal buffer from the stack or a localized region, bypassing the global heap entirely [31].

## 4.2 Rust and the Ownership Model (Influence on C++)

Rust is a separate language, but its ownership model has made ideas like deterministic cleanup and restricted aliasing more mainstream. The basic pattern is to tie a resource to a value and to free it automatically when that value goes out of scope (a common C++ term for this is RAII: acquire in a constructor, release in a destructor).

**Ownership.** Every value has a single owner. When the owner goes out of scope, the memory is dropped. This is deterministic destruction.

**Borrowing.** References can be shared (read-only) or exclusive (read/write). Rust’s compiler (the “borrow checker”) enforces these rules at compile time, which helps prevent data races without runtime locks [32].

**C++ Adoption.** Modern C++ (post-C++11) emulates this via `std::unique_ptr` (sole ownership) and `std::shared_ptr` (reference counting). Move semantics (`std::move`) allow ownership to be transferred between scopes without copying data. However, unlike Rust, C++ cannot statically prevent “use-after-move” or dangling references in all cases; it relies on developer discipline and static analysis tools [33]. Libraries like `rusty.hpp` attempt to backport Rust’s stricter semantics to C++, but the primary method is the disciplined use of smart pointers.

## 4.3 Object Pooling

When predictable response time matters and a tracing GC cannot be removed, some systems reuse objects instead of constantly allocating new ones. This pattern is called *object pooling*.

**Pattern.** A fixed set of objects is created at startup. When an object is needed, it is borrowed from the pool; when done, it is returned.

**Latency Impact.** By reusing objects, the application can generate little to no garbage in the steady state, reducing the chance of long GC pauses (sometimes called “stop-the-world” collections, because the program is briefly paused).

**Challenges.** Object pooling introduces several complexities: (1) *Memory Leaks*—if an object is not returned to the pool, the resource is lost; (2) *State Contamination*—if an object is not properly reset (cleared) before reuse, data from a previous transaction might leak into the new one; (3) *Concurrency*—accessing

a shared pool requires synchronization. Modern implementations use lock-free data structures (like Michael-Scott queues) or thread-local pools to minimize contention [34].

**Java Evolution.** Modern JVMs allocate small objects quickly (often using per-thread allocation buffers, or TLABs), so pooling is not always a win. It can still help for large objects or when the goal is to reduce worst-case pauses by limiting how much garbage is created [35].

## 5 Arbitrary-Precision Arithmetic

Fixed-width integers (such as 32-bit and 64-bit) can overflow once they exceed the maximum value the type can represent. This section surveys arbitrary-precision integers (often called big integers or bignums), which represent an integer using as many machine words as needed, limited mainly by available memory. Big integers are widely used in public-key cryptography, computer algebra systems, and high-precision computation.

### 5.1 Internal Memory Representation

The efficiency of big-integer arithmetic depends heavily on how the number is stored in memory. Most implementations represent a large integer as a sequence of fixed-size chunks of bits (often called *limbs*). Conceptually, this is equivalent to writing the number as a polynomial  $P(B) = d_n B^n + \dots + d_1 B^1 + d_0$ , where  $B$  is the base and  $d_i$  are limb values.

**Python: Base  $2^{30}$  Digits.** Python (post-version 3) represents all integers as bignums using the `PyLongObject` structure.

**The `ob_digit` Array.** The integer is stored as an array of “digits” in base  $2^{30}$  (on 64-bit systems) or  $2^{15}$  (on 32-bit systems).

**Why 30 bits?** A 64-bit CPU can perform arithmetic on 30-bit numbers and store the intermediate result (up to 60 bits) in a standard 64-bit register without overflow. This leaves bits for carry accumulation, simplifying the implementation of addition and multiplication loops in C.

**Structure.**

```
struct _longobject {
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
    Py_ssize_t ob_size; // Sign is stored here (+/- length)
    digit ob_digit[]; // Variable length array
};
```

The `ob_size` field indicates both the number of digits and the sign (negative size implies a negative number). Encoding the sign in the length field saves space and keeps the object layout compact [36].

**Java: BigInteger Magnitude.** Java’s `java.math.BigInteger` uses a base  $2^{32}$  representation.

**Representation.** A sign-magnitude representation.

```
int signum; // -1, 0, or 1
int[] mag; // Big-endian binary representation
```

**Limb Size.** It uses 32-bit `int` chunks (limbs). Unlike Python, it uses the full 32 bits. Since Java `int` is signed, operations must often cast to `long` (64-bit) to handle carries correctly. The `mag` array is stored in big-endian order (most significant int first), which makes conversion to and from big-endian byte arrays straightforward [38].

## 5.2 Multiplication Algorithms

Multiplication is typically the most expensive basic operation on big integers. The naive “schoolbook” method takes  $O(N^2)$  time for  $N$  limbs, so libraries switch to faster methods as numbers grow. The exact thresholds vary by implementation.

**Karatsuba Algorithm ( $O(N^{1.585})$ ).** The Karatsuba algorithm is a divide-and-conquer approach. It splits two  $N$ -digit numbers  $X$  and  $Y$  into high and low halves:

$$X = X_1 B^m + X_0 \tag{1}$$

$$Y = Y_1 B^m + Y_0 \tag{2}$$

Naive multiplication requires 4 sub-multiplications ( $X_1 Y_1, X_1 Y_0, X_0 Y_1, X_0 Y_0$ ). Karatsuba requires only 3: (1)  $A = X_1 \cdot Y_1$ ; (2)  $C = X_0 \cdot Y_0$ ; (3)  $B = (X_1 + X_0) \cdot (Y_1 + Y_0) - A - C$ . The result is  $AB^{2m} + BB^m + C$ . This reduction improves complexity to  $N^{\log_2 3} \approx N^{1.585}$ . It is typically used for numbers ranging from roughly 20 to 2,000 limbs. Recursion depth is limited by the overhead of additions; implementations switch back to schoolbook multiplication for small leaves [39].

**Toom-Cook Multiplication ( $O(N^{1.465})$  for Toom-3).** Toom-Cook generalizes Karatsuba by splitting the inputs into  $k$  parts. Toom-3 (where  $k = 3$ ) splits inputs into 3 parts, treating them as quadratic polynomials.

**Process.** The algorithm proceeds in three phases: (1) *Evaluation*—evaluate the polynomials at fixed points (typically  $0, 1, -1, 2, \infty$ ), transforming the multiplication problem into pointwise multiplication of values; (2) *Pointwise Multiplication*—multiply the values at these points; (3) *Interpolation*—solve a linear system to reconstruct the product polynomial from the point values.

Toom-3 reduces 9 multiplications to 5, achieving complexity  $O(N^{\log_3 5}) \approx O(N^{1.465})$ . It typically outperforms Karatsuba for numbers larger than a few thousand bits. The choice of points  $(0, 1, -1 \dots)$  is critical to ensure the inversion matrix involves only small constants (powers of 2) that can be computed via bit shifts [41].

**Table 1.** Comparative analysis of arbitrary-precision arithmetic implementations.

Aspect	Python	Java	C++ (GMP)	Haskell
Representation	Base $2^{30}$	Base $2^{32}$	64-bit limbs	GMP backend
Schoolbook	Small $N$	Small $N$	Very small $N$	Small $N$
Karatsuba	$\sim 70$ digits	$\sim 80$ ints	Tuned	Via GMP
Toom-Cook	Not typical	$\sim 240$ ints	Toom-3,4,...	Via GMP
FFT-based	Not typical	Not standard	Huge $N$	Via GMP

**Schönhage-Strassen Algorithm ( $O(N \log N \log \log N)$ ).** For extremely large integers (tens of thousands of digits), libraries often use FFT-based multiplication. The Schönhage-Strassen algorithm uses a variant of the Fast Fourier Transform (FFT) to compute the convolution of digit blocks efficiently.

**Convolution Idea.** Polynomial multiplication can be reduced to a convolution of coefficients, which can be computed by transforming to a frequency-like domain, multiplying pointwise, and transforming back:  $A(x) \cdot B(x) = \text{IFFT}(\text{FFT}(A) \cdot \text{FFT}(B))$ .

**Number Theoretic Transform (NTT).** Instead of using complex numbers (which can introduce rounding error), implementations use an FFT-like transform over integers modulo special numbers (for example, Fermat numbers of the form  $2^{2^k} + 1$ ). This keeps the results exact [43].

**Recursive Structure.** The pointwise multiplications in the transform domain are themselves large integer multiplications, solved recursively. This yields the  $O(N \log N \log \log N)$  complexity [43].

**Practical Use.** Java’s `BigInteger` switches to Karatsuba (at  $\sim 80$  ints) and Toom-Cook-3 (at  $\sim 240$  ints). Libraries like GMP (used by Haskell) use FFT-based methods for very large operands [45].

### 5.3 Comparison of Implementations

Table 1 presents a comparative analysis of arbitrary-precision arithmetic implementations across major languages.

## 6 Conclusion

This survey compares three areas that often appear together in modern systems: using types to express value constraints, managing memory with predictable costs, and implementing big integers efficiently. Across languages, similar trade-offs repeat: stronger guarantees often shift work to compile time or to tooling; more predictable memory behavior usually requires more explicit thinking about lifetimes; and efficient arbitrary-precision arithmetic depends on careful representation choices and algorithm switching.

The details differ across ecosystems, but the underlying ideas recur. Understanding them helps readers interpret safety and performance claims more

critically, and it provides a clearer set of tools to reach for when a project needs stricter correctness guarantees or more predictable performance.

## References

1. Subrange Types - Schneider Electric, accessed on December 12, 2025. [https://product-help.schneider-electric.com/Machine%20Expert/V1.1/en/SoMProg/SoMProg/Data\\_Types/Data\\_Types-14.htm](https://product-help.schneider-electric.com/Machine%20Expert/V1.1/en/SoMProg/SoMProg/Data_Types/Data_Types-14.htm)
2. Refinement Types For Haskell - Simon Peyton Jones, accessed on December 12, 2025. <https://simon.peytonjones.org/assets/pdfs/refinement-types-for-haskell.pdf>
3. Refinement Types - Liquid Haskell - Niki Vazou, accessed on December 12, 2025. [https://nikivazou.github.io/lh-course/Lecture\\_01\\_RefinementTypes.html](https://nikivazou.github.io/lh-course/Lecture_01_RefinementTypes.html)
4. Refinement Types 101 - LiquidHaskell, accessed on December 12, 2025. <https://goto.ucsd.edu/~gridaphobe/liquid/haskell/blog/blog/2013/01/01/refinement-types-101.lhs/>
5. Refinement-Types Driven Development: A study - arXiv, accessed on December 12, 2025. <https://arxiv.org/html/2509.15005v1>
6. Toward Hole-Driven Development in Liquid Haskell, accessed on December 12, 2025. <https://users.soe.ucsc.edu/~lkuper/papers/typed-holes-lh-hatra21.pdf>
7. Custom pluggable types for Java - The Checker Framework Manual, accessed on December 12, 2025. <https://checkerframework.org/manual/>
8. Building and Using Pluggable Type-Checkers | Request PDF - ResearchGate, accessed on December 12, 2025. [https://www.researchgate.net/publication/221555564\\_Building\\_and\\_Using\\_Pluggable\\_Type-Checkers](https://www.researchgate.net/publication/221555564_Building_and_Using_Pluggable_Type-Checkers)
9. davidstone/bounded-integer: C++ library aiming to replace all built-in integers - GitHub, accessed on December 12, 2025. <https://github.com/davidstone/bounded-integer>
10. cbi - compile-time bounded integers : r/cpp - Reddit, accessed on December 12, 2025. [https://www.reddit.com/r/cpp/comments/rye7p5/cbi\\_compiletime\\_bounded\\_integers/](https://www.reddit.com/r/cpp/comments/rye7p5/cbi_compiletime_bounded_integers/)
11. C++26 Contract Assertions, Reasserted - Open-std.org, accessed on December 12, 2025. <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3846r0.pdf>
12. What's new in C++26: contracts (part 3), accessed on December 12, 2025. <https://mariusbancila.ro/blog/2025/03/29/whats-new-in-cpp26-contracts-part-3/>
13. Contracts for C++ explained in 5 minutes - timur.audio, accessed on December 12, 2025. [https://timur.audio/contracts\\_explained\\_in\\_5\\_mins](https://timur.audio/contracts_explained_in_5_mins)
14. typing — Support for type hints — Python 3.9.24 documentation, accessed on December 12, 2025. <https://docs.python.org/3.9/library/typing.html>
15. Python Types Intro - FastAPI, accessed on December 12, 2025. <https://fastapi.tiangolo.com/python-types/>
16. WIT Reference - The WebAssembly Component Model, accessed on December 12, 2025. <https://component-model.bytecodealliance.org/design/wit.html>
17. Types — WebAssembly 3.0 (2025-11-22), accessed on December 12, 2025. <https://webassembly.github.io/spec/core/valid/types.html>
18. Bringing the Web Up to Speed with WebAssembly - Communications of the ACM, accessed on December 12, 2025. <https://cacm.acm.org/research/bringing-the-web-up-to-speed-with-webassembly/>

19. Security - WebAssembly, accessed on December 12, 2025. <https://webassembly.org/docs/security/>
20. Generating Stack Machine Code Using LLVM - GitHub, accessed on December 12, 2025. [https://raw.githubusercontent.com/wiki/etclabscore/evm-llvm/files/LLVM\\_talk.pdf](https://raw.githubusercontent.com/wiki/etclabscore/evm-llvm/files/LLVM_talk.pdf)
21. Register Allocation Deconstructed - LLVM, accessed on December 12, 2025. <https://llvm.org/pubs/2009-04-SCOPES-RegisterAllocationDeconstructed.pdf>
22. Greedy Register Allocation in LLVM 3.0 - The LLVM Project Blog, accessed on December 12, 2025. <https://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html>
23. Secure Register Allocation for Trusted Code Generation - UF CISE, accessed on December 12, 2025. <https://www.cise.ufl.edu/research/cad/Publications/esl22.pdf>
24. Maximizing Performance in C++ with Return Value Optimization (RVO) | by Harshit Choukse, accessed on December 12, 2025. <https://medium.com/@harshit.choukse012/maximizing-performance-in-c-with-return-value-optimization-rvo-52f0a396d3c0>
25. Introduction to Copy Elision and Return Value Optimization in C++ - Pass4sure, accessed on December 12, 2025. <https://www.pass4sure.com/blog/introduction-to-copy-elision-and-return-value-optimization-in-c/>
26. Escape analysis - Wikipedia, accessed on December 12, 2025. [https://en.wikipedia.org/wiki/Escape\\_analysis](https://en.wikipedia.org/wiki/Escape_analysis)
27. The Purpose and Mechanics of Escape Analysis in the JVM - Medium, accessed on December 12, 2025. <https://medium.com/@AlexanderObregon/the-purpose-and-mechanics-of-escape-analysis-in-the-jvm-f02c17860b8c>
28. Escape! From the Allocations: Escape Analysis in Pypy, LuaJIT, V8, C++, Go and More, accessed on December 12, 2025. <https://kippl.ly/escape-analysis/>
29. High Performance Memory Management: Arena Allocators | by Ng Song Guan - Medium, accessed on December 12, 2025. <https://medium.com/@sgn00/high-performance-memory-management-arena-allocators-c685c81ee338>
30. Region-based memory management - Wikipedia, accessed on December 12, 2025. [https://en.wikipedia.org/wiki/Region-based\\_memory\\_management](https://en.wikipedia.org/wiki/Region-based_memory_management)
31. C++ Arena Allocation Guide | Protocol Buffers Documentation, accessed on December 12, 2025. <https://protobuf.dev/reference/cpp/arenas/>
32. Rust Ownership, Borrowing & Lifetimes Explained (2025): The Core Concepts | by Ali Aslam, accessed on December 12, 2025. <https://medium.com/@aiguy/rust-ownership-borrowing-lifetimes-explained-2025-rusts-secret-sauce-b3e98634f19b>
33. Short intro to C++ for Rust developers: Ownership and Borrowing - Ironi Blog, accessed on December 12, 2025. <http://nercury.github.io/c++/intro/2017/01/22/cpp-for-rust-devs.html>
34. A Well Known But Forgotten Trick: Object Pooling - High Scalability -, accessed on December 12, 2025. <https://highscalability.com/a-well-known-but-forgotten-trick-object-pooling/>
35. Pulling ahead with Object Pooling - Data Intellect, accessed on December 12, 2025. <https://dataintellect.com/blog/pulling-ahead-with-object-pooling/>
36. Python internals: Arbitrary-precision integer implementation | Artem ..., accessed on December 12, 2025. <https://rushter.com/blog/python-integer-implementation/>
37. How Python Handles Gigantic Integers - Arpit Bhayani, accessed on December 12, 2025. <https://arpitbhayani.me/blogs/long-integers-python/>

38. Guide to Java BigInteger | Baeldung, accessed on December 12, 2025. <https://www.baeldung.com/java-biginteger>
39. Karatsuba algorithm for fast multiplication using Divide and Conquer algorithm - GeeksforGeeks, accessed on December 12, 2025. <https://www.geeksforgeeks.org/dsa/karatsuba-algorithm-for-fast-multiplication-using-divide-and-conquer-algorithm/>
40. Karatsuba Multiplication Algorithm | by Sachin Gupta - Medium, accessed on December 12, 2025. <https://medium.com/@sachinkg12/karatsuba-multiplication-algorithm-f60c4abe5779>
41. Toom Cook method for multiplication - OpenGenus IQ, accessed on December 12, 2025. <https://iq.opengenus.org/toom-cook-algorithm-multiplication/>
42. Toom–Cook multiplication - Wikipedia, accessed on December 12, 2025. [https://en.wikipedia.org/wiki/Toom%E2%80%93Cook\\_multiplication](https://en.wikipedia.org/wiki/Toom%E2%80%93Cook_multiplication)
43. Schönhage–Strassen algorithm - Wikipedia, accessed on December 12, 2025. [https://en.wikipedia.org/wiki/Sch%C3%B6nhage%E2%80%93Strassen\\_algorithm](https://en.wikipedia.org/wiki/Sch%C3%B6nhage%E2%80%93Strassen_algorithm)
44. schönhage–strassen multiplication, accessed on December 12, 2025. <http://simonrs.com/eulercircle/crypto2019/albert-ss.pdf>
45. BigInteger (Java Platform SE 8 ) - Oracle Help Center, accessed on December 12, 2025. <https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html>
46. Fast Integer Multiplication with Schönhage-Strassen's Algorithm - Inria, accessed on December 12, 2025. <https://algo.inria.fr/seminars/sem08-09/kruppa-slides.pdf>
47. A WASM-Subset Stack Architecture for Low-cost FPGAs using Open-Source EDA Flows, accessed on December 12, 2025. <https://arxiv.org/html/2512.00974v1>
48. Java BigInteger: A Comprehensive Guide | by Malinda Gamage | Medium, accessed on December 12, 2025. <https://medium.com/@pkgmalinda/java-biginteger-a-comprehensive-guide-3231f17ce27d>
49. Contract assertions (since C++26) - cppreference.com, accessed on December 12, 2025. <https://en.cppreference.com/w/cpp/language/contracts.html>
50. Thanks. What is latency hiding? - Hacker News, accessed on December 12, 2025. <https://news.ycombinator.com/item?id=26444397>
51. How to understand the "hide latency" - CUDA Programming and Performance, accessed on December 12, 2025. <https://forums.developer.nvidia.com/t/how-to-understand-the-hide-latency/258938>
52. The Art of Latency Hiding in Modern Database Engines - VLDB Endowment, accessed on December 12, 2025. <https://www.vldb.org/pvldb/vol17/p577-huang.pdf>