Symbolic Algebra in Standard C++ Verified at Compile Time*

Vincent Kräutler

November 7, 2026

Abstract

This paper describes a symbolic-algebra library in a mainstream systems programming language: standard C++23 (the category **Cpp**), capable of using compile-time evaluation in part or in full as a form of static program analysis, verification and optimization. The initial step is a deliberate restriction to a fragment of C++23 which is both a) referentially transparent (**Per**, via `const`, `constexpr`, `std::semiregular`) and b) structurally typed by default (syntactic type constraint satisfaction and template specialization via `concept`). This fragment is named the *Juliet Posture* (**Jlt** = **Per** ∩ **Cpp**) [3]: just like in dynamic duck typing, a type is known by its structure, not its name. The resulting code has a deliberate, sweet and flowery smell: **Jlt** admits a System F reading of the C++23 type system. On this foundation an embedded, lazy domain-specific language is built which supports

1. a logic lattice of truth-values (**Pst** = **Jlt**∩**Chain**): the bounded chains $\mathbb{B}$, $\mathrm{K}_3, \ldots$ (Post's many-valued logics) that classify membership,
2. a set-comprehension engine (**Lwv** = **Jlt** ∩ **Set**), valued in **Pst**, whose sets are defined in an intensional way: via a membership test ($\{x \in S \mid P(x)\}$, `S & P(x)`) rather than by listing elements,
3. a point-free relational calculus (**Trsk** = **Lwv**∩**Rel**) built on cartesian products ($\{(x, y) \in S \times S \mid P(x, y)\}$, `S * S & P(x, y)`), and
4. symbolic algebra **Ddk** = **Trsk** ∩ **Alg** built on operations over the canonical sets $\mathbb{B}, \mathbb{N}, \ldots$ ($(x, y, z) \in S^3 \mid z = f(x, y)$, `S * S * S & z == f(x, y)`).

The embedded domain-specific language spells out the corresponding algebraic laws as types, enabling the compiler to verify and optimize the expression tree. The approach should port to other systems programming languages with a rich enough type system, specifically Rust.

*Draft intended for submission to *Programming* [1], the journal for programming-related research. The paper's scope is deliberately narrower than the companion project report: this document focuses on the disciplined fragment of C++23 described here and the set language built on it. The implementation is open source as `dedekind` [2]; the broader library (categorical foundations, algebra, numbers, geometry, linear algebra, morphologies, Python facade) is documented in the companion Report distributed alongside the source tree.

Contents

1 Introduction **3**

2 A Disciplined Fragment of C++: The Juliet Posture Jlt **7**

3 A Reification of Set **8**
3.1 The Logic Lattice: Truth on a Bounded Chain 8
3.2 Sets over the Lattice 11
3.3 An Algebra of Sets 12
3.4 Ordered Sets and Quantifiers 14
3.5 Finite and Extensional Sets 16

4 A Reification of RAlg **17**
4.0.1 Functions and Morphism Factorization Gates 22
4.0.2 Optimization: arg max over a Function 26

5 A Reification of Alg(−) **26**
5.1 Algebraic Sets on the Primitive Carriers 27
5.2 Mechanical Support for Advanced Carriers 30
5.3 Applicability to Linear Algebra 30

6 Discussion and Future Work **30**

A System F, Structuralism, and Jlt **33**

B Certification and Honest Rejection of Carrier Semantics: The Traits Registry **37**
B.1 Three Tiers of Certification 37
B.2 Certification and Rejection of Basic Arithmetic 38
B.3 Algebras of the Primitive Types 39

C An Elementary Theory of the Category of Sets **41**
C.1 Small Categories (`:small` partition) 41
C.2 Cartesian Categories (`:limit` partition) 42
C.3 Cartesian Closed Categories (`:cartesian` partition) 42
C.4 Elementary Topoi (`:topoi` partition) 42
C.5 Concrete Topoi (`:concrete` partition) 43
C.6 Set-Modeling Topoi (`:etcs` partition) 43

D Point-Free Morphisms: Relations and Factorization Gates **44**

E The Carrier Poset and its Realisations **45**
E.1 How the certificates lift 45

F From Cardinality Pruning to Type-Directed Collapse **48**

G The Compiler as Structural Gate and Optimization Engine **54**

H Use of Agentic Systems and AI-Assistance Disclosure **56**

Keywords: C++23; concepts; non-type template parameters; compile-time verification; Axiomatic Systems Programming; type-directed collapse.

1 Introduction

Recent breakthroughs in machine learning have resulted not just in an explosion of novel and interesting use cases but also sparked public debate about the technology's demands and bottlenecks: Hardware cost, computational cost, energy use, and interpretability are often named in applicable literature. Here, the focus is placed on a complementary approach: correct-by-construction systems engineering, where the view is that white-box modeling, interpretability and computational efficiency are facilitated by way of static program verification and optimization. To this end, an embedded domain-specific language (eDSL) was implemented in a *mainstream systems programming language* – the very substrate that these large models build on. The resulting `dedekind` library targets analytics and machine-learning use cases; it moves correctness checks and optimization to build time, so mismatches surface as compile errors rather than runtime faults. It rests on two pillars:

1. The domain is grounded in category theory, transitioning from morphisms and small categories to cartesian-closed categories, the elementary theory of the category of sets (ETCS) and finally universal algebra [4, 5]. This framework provides pre-validated axioms, theorems and terminology and allows a minimalist type checker to drive optimizations via equivalence relations.

2. On the encoding side, the choice was more open, as any sufficiently expressive type system was considered to be a plausible implementation substrate. C++23 [6] was selected as it met the following criteria: a) it should be mainstream so that tooling is provided for; b) its default runtime requirements should be modest so as not to preclude embeddings in higher-level languages or deployment on constrained environments; c) its type system should be rich enough to perform many of the desired type checks.

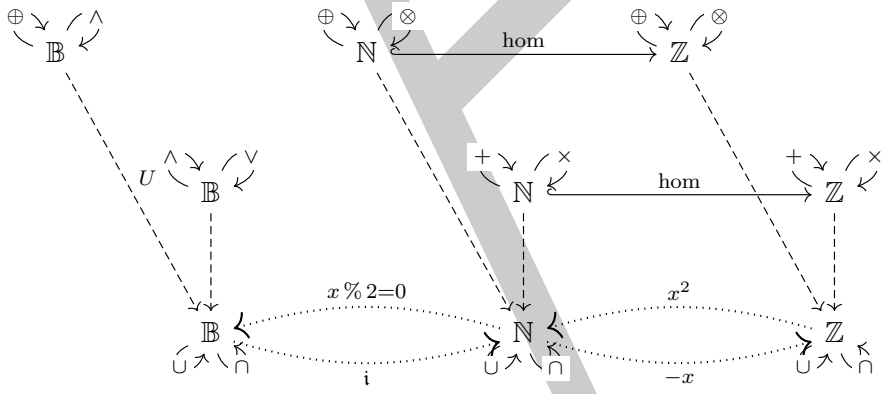


Figure 1: A sketch of the domain the paper builds on: familiar carriers and their operations, packaged as algebraic sets. The bottom row holds example carrier sets $\mathbb{B}, \mathbb{N}, \mathbb{Z}$, the objects of **Set**, each carrying its set operations: union $\cup$ and intersection $\cap$ are drawn as loops, while complement $\complement$ and the power object $\mathcal{P}$ complete the Boolean algebra of sets but are left off the diagram. Above them, staggered across two tiers, sit algebraic sets, the objects of $\mathbf{Alg}(\Sigma)$: each carrier decorated with its set element operations as loops, from the Boolean lattice and Galois field $\mathbb{F}_2$ on $\mathbb{B}$ to ordinary $(+, \times)$ and tropical $(\oplus, \otimes) = (\max, +)$ arithmetic on $\mathbb{N}, \mathbb{Z}$. The dashed forgetful functor U recovers the bare carrier; the solid arrows preserve structure, the dotted ones do not. The subsequent sections develop the reading.

Figure 1 sketches the domain of the first pillar. It is deliberately vanilla: carrier sets such as $\mathbb{B}, \mathbb{N}, \mathbb{Z}$ with the arithmetic and set operations of elementary mathematics, packaged as algebraic structures. Throughout, a *carrier* is such an underlying set (the bottom row of Figure 1), and an *algebraic set* is a carrier decorated with operations (a tier above it). The rules are not in doubt, being pre-validated and universally agreed, yet only very little is reified and programmatically accessible: a C++ operator or method signature may carry closure, but only little in terms of associativity, no distributivity, no units and no inverses. Supplying that knowledge once, and letting the compiler enforce it thereafter, is what turns these familiar rules into build-time checks and optimizations; Appendix B records which laws the type system can and cannot certify, and Appendix E catalogues them across the poset of carriers.



Figure 2: The target operational model positioning `dedekind` as a faithful intermediary between **Cat** (mathematical structure in general, axiomatically defined) and a fragment of **Cpp** (the category of C++ types, values and arrows between them). The steps of a single development cycle are: 1. find pre-validated *existing proofs* and turn them into specifications, 2. translate the specifications into source code and tests, 3. type-check the code and run the tests, reviewing the output, 4. compare and contrast with the original textbook specification (1.).

The cycle of Figure 2 runs in one direction: a pre-validated proof is cited from **Cat**, encoded and type-checked in the **Cpp** fragment, and compared back to its source, so verified evidence accumulates and the compiler can validate an imported axiom or derive further from it [7]. The posture is explicit: `dedekind` is a systems-programming library that treats pre-validated mathematical concepts as first-class compile-time entities, which keeps downstream developers from bolting fragile, ad-hoc traits onto unverified numeric types. Landing semantics at the compile-time layer, rather than as runtime abstractions or an external DSL, is the structural commitment that shapes what the compiler can enforce; against a backdrop where partial automatic differentiation exists in specialised environments [8, 9], `dedekind` contributes an end-to-end realization inside unmodified standard C++23, an existence proof that a template engine can collapse many highly non-trivial algorithmic problems down to specialized, optimized LLVM primitives.

A recent shift in the cost–benefit calculus. Two recent developments are the *key enablers*, not tricks or shortcuts. First, the C++23 standard [6] delivers the key features the technique needs: `constexpr`, `concept`s, NTTPs, and named `module`s in an open source mainstream toolchain. Previous template-metaprogramming traditions had to encode these implicitly at steep readability and review cost. Second, modern coding-assistant LLMs [10] compress the time and expertise needed to do literature search and carry out the implementation and validation work. What was once a metaprogramming curiosity is now an actionable workflow for working software engineers. The correctness story is largely unchanged, as the type system and `static_assert` obligations are what the compiler discharges, both checked mechanically. It is the cost story that has shifted, and that is what increases the discipline's viability at industrial scale rather than as only a paper artifact.

Intended target. The discipline may be interesting to computational scientists or machine learning engineers looking to leverage static verification techniques to validate and optimize their models. It may also be interesting for machine learning applications targeting resource-constrained environments such as mobile phones or other embedded or edge devices. In the latter case, the technique's compile-time-substrate posture sits one layer below existing embedded ML

runtimes such as TFLite Micro [11].

The compiler as a verification engine. Clang++ is used a formal system where a successful type check serves as a structural proof against a fixed set of inference rules [7]. The C++23 type system, while limited, is sufficiently rich to name predicates, enforce structural boundaries, and license the LLVM backend to completely remove dead code paths that have been resolved at translation time. This architecture thus enforces a clear division of labor: the library user holds an out-of-band honesty obligation when registering trait axioms, and the compiler mechanically discharges every subsequent `static_assert`, `requires` clause, or `constexpr` evaluation. A simple proof search is mimicked by operator resolution.[1]

Definition 1 (Reification)**.** *To* reify *a structure is to encode it as propositions-as-types [7]: a law becomes a* `concept`*, and a carrier satisfying the* `concept` *is a proof-as-program the type checker discharges. Reification records what holds without deciding it: for an unbounded carrier the* `concept` *rests on a registered axiom (the honesty obligation), the compiler checking its consequences.*

Existential proofs, not universal ones. That division of labour is forced by a limit, not chosen for convenience. Whether an arbitrary carrier obeys a given algebraic law is a non-trivial semantic property, and so undecidable in general (Rice's theorem [14]). A universal proof would imply the existential over any inhabited domain, so the existential is the strictly weaker claim, and the one within reach: the library does not decide a law for every carrier, it names a specific carrier, a specific law, a specific reduction, and lets the compiler discharge it. Rice's theorem is thus less a wall than an invitation to contribute: it marks the seam Figure 2 operationalizes, between the aspirational **Cat** (a universal conjecture, a general automatic verifier) on the one hand and the **Cpp** fragment of current best practice (checkable existential demonstrations with mechanical support) on the other, and a plausible conjecture is precisely an invitation to search for the existential witness.[2]

Galois specialisation. The recovery has a recurring shape, a meta-algorithm we name *Galois specialisation* (Definition 2) and apply throughout.

Definition 2 (Galois specialisation)**.** *Whether an arbitrary carrier satisfies a law is undecidable in general (Rice's theorem [14]). A* Galois specialisation *discharges it on a structured sub-domain in four steps, one development cycle of Figure 2:*

1. ***A truth theorem****: the law holds over the class its axioms define.*
2. ***A computability theorem****: the law becomes decidable on the sub-domain fixed by the invariants the carrier's operations preserve. Operations and the invariants they preserve are paired by a Galois connection [16], and the same pairing settles the difficulty of a constraint problem [17], echoing Noether's link between symmetries and invariants [18]. Where less structure is at hand, a decidable fragment [19] or a sound over-approximation [20] serves instead.*

[1] Fully automated, this is Augustsson's Djinn [12], which synthesises an inhabiting term from a type, deciding inhabitation on the propositional fragment through Dyckhoff's terminating calculus [13]. The present approach is strictly weaker: it splits that single procedure across the loop of Figure 2, where a carrier is proposed and `clang` discharges it, propose-and-check in place of decide-and-synthesise, without the completeness a decision procedure would give. Djinn can synthesise precisely because it confines itself to a decidable fragment; the present setting does the opposite, admitting the undecidable region and letting the compiler *witness* decidability wherever a carrier makes it hold, rather than requiring it up front.

[2] Penrose [15] reads Gödel's incompleteness in the same spirit: a formal system closing at one level motivates the next, not as exhaustion but as a natural successor.

3. ***A `concept`*** *approximating (1) and (2) together: sound relative to the registered axioms, so at worst it under-decides, and misleads only if a registered axiom is false.*

4. ***A carrier type*** *satisfying that* ***`concept`****: a checkable proof-as-program [7] in the realizability sense [21]. It discharges the* ***`concept`****, not the law itself; an unbounded carrier still rests on the registered axiom.*

Steps (1), (3) and (4) are a reification (Definition 1); step (2), the Galois connection, is what lets it decide rather than merely record. The outer loop is Lakatos's [22]: meet a counterexample, restrict the domain by incorporating the missing lemma as a hypothesis, and retry.

Mechanical discharge is not a proof. The discharge is not itself a proof of the law: for an unbounded carrier the author registers the axiom (the honesty obligation above) and the compiler mechanically checks its consequences. This is the bargain the proof assistants already accept: Coq [23] and Lean [24] check a development against a trusted kernel and a base of accepted axioms, and their `Admitted` and `sorry` escape hatches admit as much. The difference here is only the checker, an optimising C++ compiler rather than a dedicated prover, which is why the same discipline returns optimised machine code as a by-product.

What this buys, concretely. Guided by the rule "name the structure in the type, let the compiler discharge the laws," the library systematically constructs an algebraic hierarchy entirely at compile time. From a handful of structural primitives (morphisms and small categories), `dedekind` builds a domain specific language to describe sets, a poset of numbers ($\mathbb{B}$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, …) and a point-free linear algebra layer, among other things, Programs are partially or fully evaluated at compile-time as they are brought to single static assignment form potentially resulting in operator fusion or complete reduction to a statically computed constant result. This categorical vocabulary is not decorative; intensional sets acting as predicates into a subobject classifier are the uniform engine driving both type-level structural pruning and compile-time law enforcement.

Contributions. Specifically, this paper provides:

- **The Juliet Posture Framework:** A formal characterization of a structurally typed fragment of C++23 under a System F reading, enabling type-checked semantic constraints without custom compiler extensions.
- **A Categorical Linear Algebra Architecture:** A novel method for representing linear maps as stateless topological operators over graph complexes rather than dense arrays, natively optimized for coordinate-free network systems.
- **An Empirical Software Artifact:** A fully realized, open-source library (`dedekind`) verifying the fundamental laws of arithmetic, calculus, and automatic differentiation at translation time.

Structure of the paper. The remainder of this paper substantiates these contributions by evaluating three core architectural claims: (1) sets operate as intensional type-checked rules rather than extensional memory containers; (2) algebraic computation remains purely symbolic until explicitly materialised; and (3) standard compilers can reliably function as validation and optimization engines for mathematical invariants. Section 2 isolates the disciplined fragment of C++23, the Juliet Posture. Section 3 bootstraps intensional sets as type-checked predicates and the type-directed collapse they license, and Section 4 lifts these to point-free mappings between sets. Section 5 builds the universal-algebra poset of carriers, and Section 5.3 recasts linear algebra as coordinate-free, matrix-free operators over them. Section 6 sets out the limitations and follow-up.

2 A Disciplined Fragment of C++: The Juliet Posture Jlt

The Juliet Posture (**Jlt**) is defined in a companion paper [3]; this section recalls what the present paper uses of it. **Jlt** is standard C++23 under three rules: values are `std::semiregular` and functions are `constexpr` and pure; generic code varies with a type only through that type's concepts and traits; and recursion only folds, over a term, a finite carrier or an explicit budget. A program that keeps the rules translates into System F by dictionary passing, concepts to record types, instances to records, constraints to parameters, and inherits termination and parametricity from the target; a program whose terms are constant-evaluable runs at compile time and at runtime unchanged, a boxed value being the one exception, built at runtime only in C++23. Equality in **Jlt** is observational, two closed terms of a carrier being equal when they evaluate to the same value, so no `operator==` is asked of a value; where this paper decides the equality of sets, it asks the carrier for it separately (§3.4). A carrier enters **Jlt** by a structural criterion, semiregularity, total `constexpr` operations and an image in the calculus's target, not by name: a type is known by what it does. Appendix A develops the same discipline as a System F ↔ Bourbaki structuralist bridge [25, 26].

	Bitwise	Arithmetic		
	`bool`	`int`	`unsigned`	`double`
closure	⊢	⊢	⊢	⊢
+ associativity	✓	×	✓	×
+ commutativity	✓	×	✓	×
+ identities	✓	×	✓	×
+ inverses	✓	×	✓	×
structure	$\mathbb{F}_2$ (field)	closure only	$\mathbb{Z}/2^n$ (ring)	closure only

Table 1: Algebraic certification of the built-in primitives as a feature table. Columns are the operators a type offers (in the C++ sense); rows are a cumulative ladder of algebraic laws, each rung assuming the ones above. Closure the compiler settles by itself (⊢); every rung below is semantic, so it is either certified by hand in the trait registry (✓) or absent, in which case the type is rejected at the gate (×). The bit-exact primitives `bool` and `unsigned` climb the whole ladder; `int` and `double` stall at closure in this additive $(+, \times)$ ladder, their associativity already lost to overflow and to IEEE rounding. Under the *logical* operators `bool` is only a Boolean lattice, without inverses; the field $\mathbb{F}_2$ is the bitwise reading. This $(+, \times)$ verdict is only half the story: read instead under `min`/`max` $(\wedge/\vee)$, the *integral* columns climb a different ladder (idempotence, commutativity, associativity, absorption, bounds and distributivity, with no inverses to carry over) to all-✓. Because `min`/`max` never overflow and never round, each of `bool`, `int` and `unsigned` is a bounded distributive lattice, certified via `IsBoundedLatticeC⌋ ategory` for the bounds together with `IsDistributiveLattice` / `IsOrderDistributiveLattic⌋ eOperations` for distributivity (`IsBoundedLatticeCategory<int>` and `<bool>` hold); that order reading, not the ring one, is the structure Jlt exploits. `double` stays out even here: `NaN` is order-incomparable, so `min`/`max` are not commutative on it and the bounds are integral-only. Appendix B.3 tabulates the full per-type detail.

The Syntactic vs. Semantic Split The boundary is the standard's own distinction between *satisfying* a concept, which the compiler decides, and *modelling* it, which it does not [3]; semantic properties, such as associativity or distributivity, remain the engineer's honesty obligation. The three concept tiers that grade this gap (literal operators, functor-parametric shape, and strict categorical proof) are set out in Appendix B.1. The discrepancy between the aspirational semantic model sketched in Figure 1 and the reality of IEEE numerical primitives as exposed in

Associative algebra (`bool`, $\veebar$, $\wedge$)

```
template <typename T = bool>
concept HasBitwiseOperators =
requires(T a, T b) {
  { a & b } -> std::convertible_to<T>;
  { a | b } -> std::convertible_to<T>;
  { a ^ b } -> std::convertible_to<T>;
  { ~a    } -> std::convertible_to<T>;
};
static_assert(
  HasBitwiseOperators<bool>);
static_assert(
  IsGaloisField<
    bool,
    std::bit_xor<bool>,
    std::bit_and<bool>>);
```

Non-associative algebra (`int`, `+`)

```
template <typename T>
concept HasRingOperators =
requires(T a, T b) {
  { a + b  } -> std::same_as<T>;
  { a - b  } -> std::same_as<T>;
  { -a     } -> std::same_as<T>;
  { a * b  } -> std::same_as<T>;
};
static_assert(
  HasRingOperators<int>);
static_assert(
  !IsAssociative<
    int,
    std::plus<int>>);
```

Listing 1: Algebraic certification of the built-in primitives as type-checked code. The `concept`s resolve the certifications listed in Table 1 and can be applied regardless of provenance or lineage. Left panel: case study of the primitive `bool` type which is an exact model of the Galois field $\mathbb{F}_2$ over $\mathbb{B}$. Right panel: case study of the primitive `int` type which is often used to mimick the ring $\mathbb{Z}$.

C++23 is highlighted in Table 1. Here, **Jlt** defaults to an *Honest Rejection Policy* which asserts that if neither mechanical nor manual certification is available at compile-time, then a violation of the corresponding axioms is assumed and the type is rejected at the gate. For example, `int` satisfies the syntactic operators for a ring, but signed-overflow undefined behavior defeats associativity. Conversely, `bool` is safely accepted as the Galois Field $\mathbb{F}_2$ because its bit-perfect operations never overflow or round, as demonstrated in Listing 1. This design mimicks the use of the `sorry` macro in Lean or the `Admitted` command in Coq, where a proof obligation is bypassed by fiat to allow downstream development.

By establishing this pure, disciplined substrate first, we gain an environment where types represent sets and values represent elements. In category theory terms, we can write **Jlt** $\subset$ **Cpp**. This forms the exact Cartesian Closed Category needed to bootstrap formal set comprehension and universal algebra in the following sections.

3 A Reification of Set

With a deterministic, structurally-typed substrate established in Section 2, mathematical sets can be bootstrapped inside the type system. This section is the Galois specialisation (Definition 2) of **Set**, the category of sets. Figure 1 in the introduction previews the functionality this section builds.

3.1 The Logic Lattice: Truth on a Bounded Chain

Sets will be indexed by truth-values, so we fix those first — algebraically, before any set is defined, to keep the construction non-circular. An *answer* is a value in a *bounded chain* Ω: a totally ordered set $\bot < \cdots < \top$ with $\wedge = \min$, $\vee = \max$, and a negation $\neg$ that is the order-reversing *reflection* about the centre. A chain is fixed by enumeration alone, and its lattice operations follow from the order, so nothing set-theoretic is presupposed (Listing 2). Two instances carry the whole paper:

```
static_assert(IsPst<bool>);
using B = Boole;
static_assert(
   (false < true)
&& (false && true) == B::AND(false,
↪  true)
&& B::AND(false, true) ==
↪  std::min(false, true)
&& B::OR(false, true) ==
↪  std::max(false, true)
&& (!true) == B::RFL(true)
&& B::RFL(false) == true
);
```

```
static_assert(IsPst<Ternary>);
using K = Kleene; using enum Ternary;
static_assert(
   (False <= Unknown) && (Unknown <=
   ↪  True)
&& K::AND(Unknown, True)  == Unknown
&& K::AND(Unknown, False) == False
&& K::OR(Unknown, False)  == Unknown
&& K::OR(Unknown, True)   == True
&& K::RFL(Unknown) == Unknown
);
```

```
Ω     ::= <any IsPst>
truth ::= <a value of Ω>       (⊥, ⊤; U on
↪  K3)
logic ::= truth
  | logic meet logic
  | logic join logic
  | refl logic
```

```
meet  ::= '&&' | 'AND' | 'min'
join  ::= '||' | 'OR'  | 'max'
refl  ::= '!'  | 'RFL'
order ::= '<=' | '<⇒'
```

Listing 2: **The answer-chain** Ω, a bounded chain (Eqn. 1): `IsPst` is a truth-object (`IsΩ`) whose values are totally ordered. **Top left (𝔹):** the native operators, the species API `AND/OR/RFL`, and the lattice min / max are one and the same operation; $\wedge/\vee = \min/\max$ on $\bot < \top$, and $\neg =$ `RFL` is the reflection, a genuine complement only here. **Top right (K_3):** the undecided U under the order. $\bot \le U \le \top$; U met with the *identity* ($\top$ for $\wedge$, $\bot$ for $\vee$) stays U, but with the *annihilator* ($\bot$ for $\wedge$, $\top$ for $\vee$) it collapses to a decided value ($U \wedge \bot = \bot$, $U \vee \top = \top$), so a $\forall/\exists$ fold stays undecided until a decisive endpoint appears; the order $\le$ (the meet-order $a \le b \Leftrightarrow a \wedge b = a$) later reifies as the subset test; and `RFL` fixes the centre ($\neg U = U$). **Bottom:** the grammar of truth expressions, each operator spelled for `bool` and for the general chain; the structure of the two panels is described above, so the code carries no inline commentary.

- $\mathbb{B} = \{\bot < \top\}$, the two-chain. Here $\neg$ swaps the endpoints and *is* a complement, so the law of the excluded middle holds: $\mathbb{B}$ is the Boolean archetype.
- $K_3 = \{\bot < U < \top\}$, the three-chain (Kleene [27]). Now $\neg$ reflects about the interior point, $\neg U = U$, which is therefore *not* a complement ($U \wedge \neg U = U \neq \bot$); the excluded middle fails and U is an *undecided* answer.

We need no more. Every bounded chain with reflection is automatically a distributive lattice and a *Kleene algebra*: the condition $a \wedge \neg a \le b \vee \neg b$ holds on any chain, since $\min(a, \neg a)$ sits at or below the centre and $\max(b, \neg b)$ at or above it. So the De Morgan/Kleene structure is a *theorem* of "chain + reflection", not a separate axiom.[3]

These truth-objects, the **Jlt** types that are bounded chains, form the logic layer

$$\textbf{Pst} := \textbf{Jlt} \cap \textbf{Chain}, \tag{1}$$

named for Post's many-valued logics [28]; the sets of §3.2 are indexed by them. Writing $A = \{x \mid P_A(x)\}$ and $B = \{x \mid P_B(x)\}$, the pointwise lifts of $\wedge, \vee, \neg$ to predicates are the set operations

[3] **Pst** is the bounded chains selectable by a type constraint alone (the chain *laws* are witnessed on each carrier, Listing 2, not certified by the constraint): $\mathbb{B}$, K_3, a Łukasiewicz $Ł_n$, and, on a purpose-built carrier, the unit interval $[0, 1]$ under min / max, the fuzzy/Gödel reading (distinct from probability's product t-norm, $a \cdot b \neq \min$). Order alone makes the chain well-behaved (§3.4), so finiteness is not required. What is excluded is the non-chain many-valued logics (a four-element Boolean algebra, Belnap's bilattice): they are not totally ordered.

(Table 2):

$$x \in A = P_A(x), \qquad \complement A = \{x \mid \neg P_A(x)\},$$
$$A \cap B = \{x \mid P_A(x) \wedge P_B(x)\}, A \cup B = \{x \mid P_A(x) \vee P_B(x)\}, \tag{2}$$
$$A = f^{-1}(B) \;\Rightarrow\; P_A = P_B \circ f \quad (f \text{ total}).$$

The answer-chain Ω is set by the logic species; the honest default is Kleene's three-valued $\{\top, \bot, U\}$ [27], so $\wedge$ is min, $\vee$ is max, and $\neg$ the flip. A set is *decidable* exactly when membership never returns U, i.e. its characteristic map $\chi_A = P_A$ factors through the *decided* sub-chain $\Sigma = \{\top, \bot\}$, the endpoints (the limits) of Ω:

$$\chi_A \;=\; \iota \circ \bar{\chi}, \qquad \bar{\chi} : A \to \Sigma = \{\top, \bot\}, \qquad \iota : \Sigma \hookrightarrow \Omega. \tag{3}$$

Strict ETCS is the Boolean collapse $\Sigma = \Omega$ ($\iota = \mathrm{id}$, so every test factors and all membership is decidable); the honest species keeps ι a *proper* inclusion $\{\top, \bot\} \subsetneq \{\top, \bot, U\}$, so a χ_A that hits U has no factorisation.[4] The factorisation $\chi_A = \iota \circ \bar{\chi}$ is the classifier-side instance of the morphism factorisation that §4.0.1 reifies as a first-class (epic-monic) structure. This is the *value* axis of (un)decidability, fixed by the chain Ω; the orthogonal *quantifier* axis (Equation 5, echoing the Chomsky hierarchy) is a property of the *domain*, and arrives with the sets below.

Table 2 reads these operations across five languages, from the Boolean archetype $\mathbb{B}$ down to the two executable-C++ rows, **Pst** (the chain itself) and **Lwv** (the sets of §3.2 built over it), placed side by side.

	member	meet	join	negation	bottom	top	order
$\mathbb{B}$	$\models$	$\wedge$	$\vee$	$\neg$	$\bot$	$\top$	$\Rightarrow$
$\mathcal{P}(A)$	$\in$	$\cap$	$\cup$	$\complement$	$\varnothing$	A	$\subseteq$
Topos	χ	$\wedge$	$\vee$	$\neg$	**0**	A	$\hookrightarrow$
Pst	`IsPst`	`&&` / `AND`	`||` / `OR`	`!` / `RFL`	`false` / `False`	`true` / `True`	`<=`
Lwv	`s(x)`	`&`	`|`	`~`	`∅`	$\mathbb{A}$	`<=`

Table 2: Operator-mapping table: each column is one construction, spelled in five languages over the rows $\mathbb{B}$, $\mathcal{P}(A)$, **Topos**, **Pst**, **Lwv**. The two executable-C++ rows are placed together at the bottom for direct comparison: **Pst** is the bounded chain (`IsPst` the truth-object type-constraint), **Lwv** the sets built over it. Each **Pst** cell is spelled twice: the native `bool` form left of the slash, and the species API (`Kleene::AND` and friends, the surface every logic species provides) right; $\neg$ is the reflection, a genuine complement only on the two-chain $\mathbb{B}$ (on K_3, $\neg U = U$). The **Topos** row is the Boolean case realised by **Set**. The *member* column is membership ($\models$ satisfaction, $\in$ set membership, χ the classifier, `IsPst` the truth-object constraint, `s(x)` the set applied to a point); the other columns are the lattice operations.

3.2 Sets over the Lattice

The chains of §3.1 already give a well-defined set: a type *is* a set, and membership reduces to the *type check* the compiler already performs, reified as the universal set `𝔸<T>` over `T`, whose characteristic map is constant-true ($\chi \equiv \top$). This much already serves the algebra and the exposition; the relational operations, subset and equality, are reified later (§3.4 onward) where

[4] A notational caution: ETCS, being Boolean, writes Ω for the two-valued object we call Σ. We reserve Ω for the *full* chain (the ambient classifier, $\{\top, \bot, U\}$ for K_3) and $\Sigma = \{\top, \bot\}$ for its endpoints (the decided sub-object), so $\Sigma \subsetneq \Omega$, never the reverse. The inclusion $\Sigma \hookrightarrow \Omega$ is a Rosolini dominance [29].

the quantifier they hide can discharge. A *proper* set adds a *predicate* to that type constraint: a carrier `A` with a characteristic map $\chi_S : A \to \Omega$ (`𝔸<A>{} | P`), whose membership is now valued in the chain and may answer U. The two layers are thus one construction at two settings of the predicate: the trivial $\chi \equiv \top$ (a bare type, the universal set) and a genuine χ (a refined subobject). Here, a set *is* its characteristic predicate, not a list of its elements. The set operations *are* the pointwise logical ones. This way, there is no problem to model an open disk $\{\, x \mid \|x - c\| < r \,\}$, which in an extensional setting is an uncountable bag no machine can hold. This intuition is formalized in Lawvere's elementary theory of the category of sets [5], a set given by objects and their morphisms. Formally speaking:

Definition 3 (ETCS Set)**. Set** is a **well-pointed**, **Cartesian Closed Category** with a **subobject classifier**, a **natural numbers object**, and the **axiom of choice** *[30].*

The DSL is grounded in it by a faithful reification: the defining axioms become type-level `concept` predicates, composed into a structural `concept IsSet` that checks whether a **Jlt** type is also an object of **Set**. The step-by-step embedding is in Appendix C, the concepts in the reference implementation [2] (Mac Lane [31] and Pierce [32] for background). Two caveats apply, and they are really one. i) The axiom of choice (Axiom 10) is only approximated through the power-object lattice, and is marked aspirational. ii) Choice is what would force the two-valued chain (a Boolean answer); `IsSet` does not assume it, so membership is valued in the full chain Ω and may answer *unknown* by default (§3.1), decidable only where the test factors through the endpoints Σ. `HasDecidableMembership` is the *sound, conservative* certificate of that factorisation, not a decision of it (checking it exactly is impossible in general, Rice [14]); admitting *unknown* by default is what keeps the grammar strictly sound.[5] With these assumptions acknowledged, one can define:

Definition 4 (**Lwv** := **Jlt** ∩ **Set**)**.** *In* **Lwv***:*

1. *A* set *over a carrier* `A` *in* **Jlt** *(*`std::semiregular`*: copies are asked of the carrier, not decidable equality, which the Boolean ETCS* `IsSet` *asks separately) is a* subobject *of* `A`*: a type* `S` *satisfying* `IsSubobject<S, A>`*, whose characteristic predicate* $\chi_S : A \to \Omega$ *classifies membership (decidably when* χ_S *factors through the Boolean* Σ*, Equation 3).*
2. *A* relation *is a subobject of the Cartesian product* `::Domain` × `::Codomain`*.*
3. *A* set *is thus just the points its test accepts: the* `true`*-slice, not the* `false` *or* `Unknown` *rest.*[6]

The full ETCS axioms live not on the individual set but on its ambient universe*: the top element* $\mathbb{A}\langle A\rangle$ *is itself an ETCS set-object, satisfying the heavier* `concept IsSet` *(Definition 3). A* **Lwv** *set is a subobject of such a universe over a* **Jlt** *carrier, so one can write:*

$$\textbf{Lwv} := \textbf{Jlt} \cap \textbf{Set}, \tag{4}$$

in which the predicate χ_S *is the set.*

With the notion of set fixed, the rest of the section develops the DSL over three tiers of increasing decidability, loosely echoing the Chomsky hierarchy [33]. Equality and subset of a general predicate (§3.3) are undecidable, Rice-walled [14]. An ordered set (§3.4) instead has a canonical normal form, its run list (the finitely many maximal intervals it is the union of), and comparison decides on those; a periodic one is *regular*, its residue the Myhill–Nerode canonical

[5] A carrier whose membership is only semi-decidable, a Mandelbrot set say, stays at the partial default; the decidable track (§3.4 onward) is the specialisation, not the assumption.

[6] Categorically, the true-fibre of χ_S: the pullback of $\top$ along χ_S, the true-slice of its graph rather than the whole graph.

form [34]. The finite, extensional sets (§3.5) simply list. Each tier's normal form is what makes its comparisons decidable.

These two undecidability sources are *orthogonal* axes, and every set operation sits at a point in their product.[7] The *dominance* axis (Equation 3) is the codomain of χ: does membership land in Σ, or hit U? The *quantifier* axis is how much of the set a query inspects. A quantifier *is* a set operation, a fold of the membership test over the domain (the structural, Bourbaki sense, taken as the definition, not a later coincidence):

$$\begin{aligned} \exists x \in S.\, P(x) &:\equiv \{x \in S \mid P(x)\} \neq \varnothing \;=\; \textstyle\bigvee_{x \in S} P(x), \qquad \forall :\equiv \neg \exists \neg; \\ \forall x \in S.\, P(x) &:\equiv \{x \in S \mid P(x)\} = S \;=\; \textstyle\bigwedge_{x \in S} P(x), \qquad \exists :\equiv \neg \forall \neg. \end{aligned} \tag{5}$$

Membership asks about one point; the quantifiers about *every* point ($\forall$) or *some* point ($\exists$).[8] The two axes now read straight off the fold: a *finite* meet or join of *decided* values is decided, so a query lands in the decidable corner *when* the predicate factors through an effective finite quotient $\sim$:

$$\underbrace{P(A) \subseteq \Sigma}_{\text{axis 1: each test decided}} \qquad \text{and} \qquad \underbrace{|S/{\sim}| < \infty}_{\text{axis 2: the fold is finite}} \quad . \tag{6}$$

This is a *sound, sufficient* certificate for the hard corner, not an exact characterisation: a direct witness or a symbolic reduction can settle a query with no finite quotient at all. Membership can be perfectly decidable ($P(A) \subseteq \Sigma$, axis 1) while the quantifier still escapes Σ, because an unbounded fold may never resolve (it stays U); a finite normal form (§3.4 onward) discharges the bound and pulls the query back into Σ (axis 2). All four combinations occur; the bottom-left is inhabited too, since $\varnothing \subseteq B$ decides even where membership in B does not. The corner is closed under finite composition (Equation 2, including the preimage line $P_{f^{-1}(B)} = P_B \circ f$), so compile-time optimisation is "stay in the corner", while Galois specialisation carves a finite normal form to "get to the corner". Deciding membership of the corner is itself undecidable (Rice [14]), so the type system recognises a *sound sub-quadrant* and answers U elsewhere.

3.3 An Algebra of Sets

Membership was fixed in §3.1: a set is its characteristic map $\chi_A : A \to \Omega$, valued in the answer-chain, and decidable exactly where it factors through the decided endpoints Σ (Equation 3). The algebra of Listing 3 adds the lattice operations and the closed `query` sort; restricting to $\Sigma = \{\top, \bot\}$ decides *membership*, while set *equality* sits on the orthogonal quantifier axis (Equation 5), decidable only where a normal form exists (§3.4 onward).

Listing 3 gives worked examples and the generic grammar of **Lwv**. A set expression (`setexpr`) builds a set from canonical or user-defined values and combines them through set operations or filtration with a predicate. A predicate (`pred`) can be assembled from an atomic equality operation, a set membership test and boolean composition rules applied to constants or set element aliases (`elem`). The cartesian `product S * T` changes the carrier ($\mathbf{Set}\langle A\rangle \times \mathbf{Set}\langle B\rangle \to \mathbf{Set}\langle A \times B\rangle$), and forms the foundation for the relational algebra discussed in §4. The `query` nonterminal defines the *closed* partial order on sets via set equality and the non-strict subset order $A \subseteq B$ (and its converse). `<any elem → 𝔹>` and `<any IsSet>` are extension points allowing for specializations such as those presented in Listing 4. Together with a pluggable `logic_spe⌋ cies` (the classifier codomain), these are the points at which further species of (un)decidability enter: the coordinate of a query is *inferred* by the compiler, not enumerated by the grammar.

[7] A set can resist decision in other ways too, e.g. an unrepresentable (uncountable) carrier or a partial map's Σ^0_1 definedness (§4); these feed the two axes here as witnesses or reduce to the quantifier one. The general setting is synthetic computability over a Rosolini dominance [29].

[8] In recursion-theoretic terms this is the arithmetic hierarchy of the query: membership atomic (Δ^0_0), the $\forall$-fold Π^0_1, the $\exists$-fold Σ^0_1.

```
constexpr auto T = 𝔹 & π == true;
constexpr auto F = ~T;
static_assert(
   T(true) && !F(true)
&& η(true)(true)
&& (F | T) == 𝔹
&& (F & T) == ∅{}
);
```

```
static_assert(
   (η(true) ^ η(false))(true)
&& (η(true) ^ η(false))(false)
&& !(η(true) ^ η(true))(true)
&& (η(true) * η(false)) ==
↪  η(std::pair{true, false})
&& (η(false) * η(true)) ==
↪  η(std::pair{false, true})
&& (𝔹 * 𝔹)(std::pair{true, false})
);
```

```
setexpr ::= generator
  | '~' setexpr
  | setexpr op2 setexpr
  | setexpr '&' pred
  | product
op2 ::= '|' | '&' | '^'
generator ::= '𝔸' carrier | '∅' carrier
  | 'η' '(' value ')'
  | <any IsSet>
carrier ::= <any std::semiregular type>
```

```
pred ::= atom
  | '!' pred
  | pred p2 pred
  | setexpr '(' elem ')'
  | <any elem → 𝔹>
p2 ::= '&&' | '||'
atom ::= elem '==' elem
  | elem '==' const
elem ::= 'π' | 'π_1' | 'π_2'
```

```
product ::= setexpr '*' setexpr
```

```
query ::= setexpr cmp setexpr
cmp  ::= '<' | '<=' | '>' | '>='  | '=='
```

Listing 3: **Generic set algebra** corresponding to Eqn. 2 and the bottom row in Table 2. **Top:** worked examples on $\mathbb{B}$: a comprehension builds a singleton, complement/union/meet run the Boolean lattice (Table 2), and membership queries the result, all on equality alone, no order on the elements. **Bottom:** the grammar in four panels, set expressions and predicates above, the cartesian product and closed queries below. A `setexpr` builds a set; a `pred` filters one; the cartesian `product` and the closed `query` sort are read out in the body. The grammar is sound; *decidability* is the separate, certified question, sort by sort (membership by the dominance of Equation 3, `query` by the tier's normal form).

By comprehension, `ℕ & π > 5` denotes $\{x \in \mathbb{N} \mid x > 5\}$, a `Halfspace`, with no wrapper (Listing 3).

The set operations are then the algebra on these predicates. Intersection (`&`) and union (`|`) compose these predicates through a generic lattice-law term reducer, by the laws of the subset order $\mathrm{Sub}(T)$ (Listing 3); complement (`~`) is handled separately, as an involution. On a chain the term is kept as written, and its normal form, the run list, is read on demand by the comparison that needs it. A meet of two intervals collapses to their greatest lower bound, an interval again; a join is a union, at most two runs. For example, the two oppositely directed halfspaces $\{x < 5\}$ and $\{x > -5\}$ are incomparable, so their meet is the bounded interval $(-5, 5)$ that is their glb, and the carrier supplies only how to compute the bound (two same-direction halfspaces are comparable, and their meet is again a halfspace). Under the classical logic a plain complement pair over a value-determined leaf, one whose value is a compile-time constant, collapses: $a \wedge \neg a$ to `∅` (non-contradiction) and $a \vee \neg a$ to the universe (excluded middle). A runtime-stateful leaf is left unreduced, since a same-type pair need not be genuine complements once values differ. A partial logic such as Kleene's K_3 declines the collapse, since $a \wedge \neg a$ can sit at the interior Unknown: its negation stays involutive ($\neg\neg A = A$) but is not a full complement.

The correspondence is exact on the decidable track, and it is what makes the algebra optimizable. The top element, the universal set A itself, is reified as `sets::𝔸<T>`; Listing 3 (left) exhibits the canonical case, the Booleans $\mathbb{B} =$ `𝔸<bool>{}`. Optimization is then the simplification of these predicate compositions at translation time: an empty meet prunes, $\bot \wedge p \Rightarrow \bot$, and constant conjuncts fold away. The induced order, set inclusion, is pointwise implication $\chi_S \Rightarrow \chi_T$; it ranges over the domain, and keeping it decidable is the task of the ordered (§3.4) and extensional (§3.5) specialisations.

The compiler is then allowed to perform aggressive compile-time verification and type-directed optimization. When a membership test or set relation is encountered, the template engine expands the composed predicates and evaluates them during compilation. If an intersection yields a logical contradiction, the type system reduces the resulting expression directly to the empty set $\emptyset$ before code generation occurs. As a direct consequence, downstream runtime membership queries for statically impossible configurations are completely eliminated by the LLVM optimizer backend, collapsing into literal constant returns within the emitted machine instructions. Listing 3 exhibits a checked-in extract that tests this type-directed collapse in practice, and a companion checked-in IR extract makes the collapse mechanically observable during the build: the witness function for set membership constant-folds to `ret i1 false` in the emitted LLVM IR.

3.4 Ordered Sets and Quantifiers

The algebra so far asks nothing of the carrier beyond copies: membership, the Boolean lattice, and the subset order $\subseteq$ — the rightmost column of Table 2 — are all pointwise lifts of the order on Ω, so they hold for any `std::semiregular` carrier, the relations of a hom-set among them; the carrier's own equality enters only with the ETCS axioms (`IsSet`). A second, independent order enters when the carrier's *elements* are themselves ordered ($\leq$): now element comparisons (`π > 5`), the halfspaces they cut by comprehension, the quantifiers over them, and the extrema below all become available. This is the first rung that asks the carrier for more than equality; its grammar extension is Listing 4.

First-order quantification is the fold of Equation 5 once more, now in code: the comprehension is the meet `S & P`, and each quantifier is one `operator==` against a bound, `exists(S,P)` being `!(∅{} == (S & P))` and `forall(S,P)` being `(S & P) == S` (Listing 4). The set comparisons are these quantifiers specialised to a membership predicate:

$$\begin{aligned} A \subseteq B &:\equiv \forall x \in A.\, B(x), & A = B &:\equiv A \subseteq B \wedge B \subseteq A, \\ A \subsetneq B &:\equiv A \subseteq B \wedge \neg(B \subseteq A). \end{aligned} \tag{7}$$

```
template <IsSet S, class P>
constexpr bool exists(S s, P p) {
  return !(∅{} == (s & p));
}
```

```
template <IsSet S, class P>
constexpr bool forall(S s, P p) {
  return (s & p) == s;
}
```

```
constexpr auto div3 = π % 3 == 0;
static_assert( exists(𝔹, π == true));
static_assert(!forall(𝔹, π == true));
static_assert( exists(ℕ, div3));
static_assert(!forall(ℕ, div3));
```

```
constexpr auto gt5 = ℕ & π > 5;
static_assert(bool(gt5(6)) &&
↪  !bool(gt5(5)));
static_assert(min(𝔹)(false));
static_assert(max(ℕ & π <= 5)(5));
static_assert(bool(℘(gt5)(gt5)));
```

set expressions | **predicates** ($\to \mathbb{B}$)

```
setexpr +::= ('max' | 'min') '(' setexpr
↪  ')'
        |  '℘' '(' setexpr ')'
```

```
pred +::= ('forall' | 'exists') '(' setexpr
↪  ',' pred ')'
atom +::= elem cmp elem
        |  elem cmp const
```

Listing 4: Ordered sets: order on the carrier's elements, the quantifiers it makes decidable, the extrema they define, and the power set; a grammar extension of Listing 3. **Predicates** (right): element comparisons `cmp` cut halfspaces by comprehension; `forall`/`exists` plug into `pred`, each one point-free `operator==` against a bound (Equation 5): `exists(S,P)` tests `S & P` against $\varnothing$, `forall(S,P)` against S. **Set expressions** (left): `max`/`min` return the extremal *subset*, defined via $\forall$ (Equation 9) and computed as `S` met with its symbolically-settled `upperbounds`/`lowerbounds`, so no candidate is enumerated. **Witnesses**: $\exists/\forall$ on the finite $\mathbb{B}$ and on $\mathbb{N}$ through the finite quotient $\mathbb{Z}/3\mathbb{Z}$ (the congruence $\pi \equiv 0 \pmod 3$); a halfspace; and extrema on $\mathbb{B}$ and $\mathbb{N}$. The power set $\mathfrak{P}(S)$ is a comprehension over the subset order, its subsets reified as the subobject $\mathrm{Sub}(C)$: $\mathfrak{P}(S) = \mathbb{A}\langle \mathrm{Sub}(C)\rangle \mid X \subseteq S$; an ordered carrier supplies $\mathrm{Sub}(C)$ and a general one does not, so $\mathfrak{P}$ is then a type error, and extensional enumeration of a finite power set is left aside. (Subset `<=` is the normal-form comparison of Listing 5.)

The power set is the predicate lift of subset: its membership test *is* the subset relation, which is why $\mathfrak{P}$ belongs to this grammar extension.

$$X \in \mathfrak{P}(S) \;:\equiv\; X \subseteq S, \qquad \mathfrak{P}(S) \;=\; \{\, X \mid X \subseteq S \,\}. \tag{8}$$

Intensionally, a quantifier collapses to one question: does the meet `S & P` reduce to always-false ($\exists$ fails) or always-true ($\forall$ holds)? The cost is then the *set relation*, not the carrier's cardinality, so $\forall$ decides even over an infinite carrier when the predicate factors through a finite quotient (the finite normal form of Equation 6): the periodic congruence $\pi \equiv 0 \pmod 3$ settles on the three residues of $\mathbb{Z}/3\mathbb{Z}$, not the countably many naturals (Listing 4). A predicate with no such factorisation (`isPrime`, a Collatz recurrence) does not reduce: the honest Rice wall, a compile error rather than a fabricated answer.

The extremum is a set operation in the same spirit: the greatest elements of S are those no element beats, so max keeps the comprehension $\forall$ tests and returns the extremal *subset*.

$$\max S \;:\equiv\; \{x \in S \mid \forall x' \in S.\; x' \le x\}, \qquad \min \;:\equiv\; \max \text{ under } \le \mapsto \ge. \tag{9}$$

In code it is S met with its symbolically-settled `upperbounds`, so no product is built and no candidate enumerated; min is the dual. The answer is a set, so the degenerate cases need no special-casing: an empty or unbounded carrier gives $\varnothing$, a total or partial order a singleton (unique by antisymmetry), a preorder a possible tie. A carrier settling no `upperbounds` matches no `max`: the same Rice wall. Section 4 lifts $\le$ to a function's graph, and the same filter becomes $\arg\max$.

3.5 Finite and Extensional Sets

The container reading and the predicate reading are almost interconvertible, and the asymmetry is the point (Listing 5). Any `std::set` is a **Lwv** set for free, its `contains` the characteristic predicate; call that embedding ι, and it is total. Only a finite, enumerable **Lwv** set materialises back to a container; call that map μ (the *extension*, dual to the predicate's intension), realised by `ext`, as η names `singleton`, the power-set monad's unit [35]; it is partial, the naturals having no finite materialisation. This asymmetry is the model/materialise gap of the whole framework: **Jlt** *models* carriers up to the continuum but *materialises* only finite ones,

$$\textbf{Jlt} \text{ models} \le \beth_1, \qquad \textbf{Jlt} \text{ materialises} < \aleph_0, \tag{10}$$

the library's modelling scope for carrier predicates, $\beth_1$ (Appendix C), above, meeting the finite reach of μ below. So **Lwv** is strictly more expressive than `std::set`, the extensional container it generalises, and the price is paid in *proof obligations*: where μ is unavailable, membership is settled by the concept constraints and factorisations of this section, not by enumeration. This is the extensional/intensional database distinction of deductive databases, `std::set` playing the extensional store (EDB, facts) and **Lwv** the intensional one (IDB, rules), with materialisation the materialised view [36].

4 A Reification of RAlg

This section is the Galois specialisation (Definition 2) of **RAlg**, the category of relations. Relations are a *dioid*, an idempotent semiring $(\mathrm{Rel}, \cup, ;, \varnothing, \Delta)$ under union and relative product, and that structure is the spine of this section (Table 3). It stands on its own two monoids: a join-semilattice for $\cup$, and a composition monoid for $;$ whose unit is the diagonal Δ. The additive half happens to be the Boolean lattice a relation already carries *as a set* (Section 3, Table 2); the multiplicative half, composition with unit the diagonal, is what the *product* adds and is the reason the section exists.

Listing 6 gives the grammar of relation expressions, built on the set expressions of Listing 3.

```cpp
std::set<int> xs{2, 3, 5};        //
↪  finite universe
auto gt3 = ℕ & π > 3;      // {x > 3}
// μ: keep the members χ accepts
↪  (runtime)
assert(ext(xs, gt3)
       == std::set<int>{5});
assert(ext(xs, ℕ & π > 5)
       == std::set<int>{});
```

materialisation μ

```cpp
// <= decides structurally, at compile
↪  time
static_assert(
  bool((ℕ & π > 5)
       <= (ℕ & π > 3)));
```

normal-form comparison

```
container ::= 'ext' '(' universe ',' pred
↪  ')'
universe  ::= <any
↪  std::ranges::input_range>
```

```
query +::= nf cmp nf   -- comparing normal
↪  forms decides cmp (⊆ order and ==)
nf     ::= 'Sub' '(' carrier ')' | residue
↪  | 'Ø' | '𝔸'
```

Listing 5: Extensional operations on sets in *normal form*: the materialisation `ext` (the partial μ, a finite universe kept where χ accepts, into a `std::set`) and the comparisons `==`/`<=`. **Top**: a finite `std::set` bounds an intensional halfspace, so their intersection always materialises even though the halfspace over $\mathbb{N}$ does not; the extensional and intensional readings compose. `ext` is a runtime retraction (it fills a `std::set`), not a `constexpr` one. **Right**: subset `<=` decides *structurally* on the normal form the carrier supplies, here two halfspaces, at compile time. **Bottom**: the grammar in two panels; the boundaries `Ø`/$\mathbb{A}$ are Listing 3's generators, and normal forms are compared by the set `query` over the single `cmp` (the subset order, Listing 3).

Definition 5 (**Trsk** := **Lwv** ∩ **Rel**)**.** *A* **Trsk** *relation is an* **Lwv** *subobject (Definition 4) of a Cartesian product $A \times B$; the* family *of such relations carries the dioid (allegory) structure of this section — converse $(\cdot)^\circ$, relative product $;$, and, on the endorelations $\mathrm{Rel}(A, A)$, the diagonal unit Δ. Where* **Lwv** *is the set fragment,* **Trsk** *is its point-free relational refinement,*

$$\textbf{Trsk} := \textbf{Lwv} \cap \textbf{Rel}, \tag{11}$$

the Tarskian calculus on which the algebra of §5 stands.

To transition seamlessly from static intensional sets to dynamic mappings, binary relations can be formalized as morphisms `IsArrow<A, B>` where both domain and codomain `require` the ETCS axioms via `IsSet`. Such morphisms are then classified by the following properties, if one focuses on the domain:

- **Totality**: Outgoing arrows per input ≥ 1.
- **Functionality**: Outgoing arrows per input ≤ 1.

And if one focuses on the codomain:

- **Injectivity**: Incoming arrows per output ≤ 1.
- **Surjectivity**: Incoming arrows per output ≥ 1.

Equivalently, relations can be represented as sets whose elements satisfy the product axioms (`IsProduct`), for example via the signature `IsSet<std::pair>`.

Table 4 organises these four properties as one structure: an *existence* bound (≥ 1) and a *uniqueness* bound (≤ 1) on each projection fibre of $R \subseteq A \times B$, holding for every base point, read across three vocabularies. The existence bound is the bounded existential $\exists$, the indexed join $\bigvee$ of Table 2 over the fibre; the uniqueness bound is the at-most-one condition

Tarski's calculus

```
relexpr ::= setexpr
          | setexpr '*' setexpr '&' relpred
          | converse '(' relexpr ')'
          | relexpr '>>' relexpr
          | diag '(' setexpr ')'
          | reflexive '(' relexpr ')'
          | symmetric '(' relexpr ')'
```

pair predicates (extends pred)

```
relpred +::= ratom
ratom    ::= term cmp term
term     ::= proj | const | term arith term
           | 'ap' '(' arrow ',' term ')'
           | 'cond' '(' relpred ',' term
           ↪  ',' term ')'
proj     ::= 'π_1' | 'π_2' | 'π_3' | 'π_<'
↪  k '>' | 'x' | 'y' | 'z'
arith    ::= '+' | '-' | '*' | '/' | '%'
```

Codd's relational algebra

```
relexpr +::= select '(' relexpr ',' relpred
↪  ')'
table    ::= join '(' relexpr ',' relexpr
↪  ')'
```

function algebra

```
setexpr +::= image '(' arrow ',' setexpr
↪  ')'
           | preimage '(' arrow ',' setexpr
           ↪  ')'
           | fibre '(' relexpr ',' elem ')'
```

spellings

```
converse  ::= 'converse' | 'conv' |
↪  'dagger'
diag      ::= 'diag' | 'Δ'
select    ::= 'select' | 'σ'
reflexive ::= 'reflexive' | 'refl'
symmetric ::= 'symmetric' | 'sym'
join      ::= 'natural_join' | 'join'
```

query

```
query +::= ('IsFunctional' | 'IsEntire')
↪  '(' relexpr ')'
arrow  ::= <an IsArrow A → B>
```

Listing 6: The grammar of **relation expressions**, a refinement of Listing 3: a relation *is* a set over a product, so complement, union `|`, meet `&` and the product `*` are *inherited* from `setexpr`. Operators are grouped by tradition and attached to their *result sort*, keeping the grammar compositional. **Tarski's calculus** (`relexpr` → `relexpr`): converse R°, relative product $R\,;S$ (`>>`, Boolean middle), the coreflexive/diagonal Δ_S (`diag`), and the `reflexive`/`symmetric` closures; a relation is also a restricted product `S * T & relpred`. **`relpred`**, the pair-predicate sub-grammar, compares terms built from the projections π_1, π_2 by arithmetic, `ap` and `cond`, its constants carried as values, and inherits `!`, `&&`, `||`, `cmp` from `pred`. It also lifts an arrow's *value* into a `relpred` atom: `y == ap(f, x)` is the point-free $y = f(x)$; forward membership needs only a decidable-equality codomain, while a *retractable* `f` (one carrying a `retract` hook) additionally makes the image decidable (its factorisation class becomes the relation's decidability class); a bare isomorphism qualifies only once it exposes that retract. This value-lift is distinct from the whole-arrow `graph` lift below. **Codd's algebra**: selection σ (`select`→ `relexpr`) and natural join $\bowtie$ (`join`→ an n-ary *table*). **Function algebra** (→ `setexpr`): image $f[S]$, pre-image $f^{-1}(S)$, and the point fibre $f^{-1}(y)$. **Spellings**: operators also accept a textbook alias where a valid identifier exists — `Δ`/`diag`, `σ`/`select` (letter-glyphs, as `π`/`η` already are); converse has none (`†` is not an identifier), so `conv`/`dagger`. The projections also take the blackboard coordinate aliases `𝕩`/`𝕪`/`𝕫` (mathematical-italic, so a valid identifier yet not ASCII `x`/`y`/`z`) for `π_1`/`π_2`/`π_3`, so a triple predicate reads as $z = f(x, y)$. **`query`** adds the closed properties `IsFunctional`/`IsEntire`; their semantics are Table 4 and Listing 7. The reflexive-transitive closure R^* is not point-free here; the whole-arrow `graph` lift (arrow → `Comprehension<𝔸<pair>, P>`) is deliberately kept out of the grammar, as it forces image and pre-image to be *inferred* — unlike the point-free value-lift `ap` above, which stays in.

	additive $\oplus = \cup$	**multiplicative** $\odot = ;$
spelling	| (union)	`>>` (relative product)
unit	$\mathbf{0} = \varnothing$ (empty)	$\mathbf{1} = \Delta = \{\pi_1{=}\pi_2\}$ (diagonal)
monoid	comm., **idempotent** (a semilattice)	assoc., **non-commutative**
involution	$(\cdot)^\circ$ converse: $(R \odot S)^\circ = S^\circ \odot R^\circ$,	$(R \oplus S)^\circ = R^\circ \oplus S^\circ$
dioid laws:	$\odot$ distributes over $\oplus$; $R \odot \varnothing = \varnothing = \varnothing \odot R$ (annihilation)	
`ext`	$\mapsto$ matrix sum $\oplus$	$\mapsto$ matrix product $\otimes$, unit I

Table 3: **Relations are a dioid** $(\mathrm{Rel}, \cup, ;, \varnothing, \Delta)$, an idempotent semiring standing on its own two monoids. The multiplicative monoid is composition $R \gg S$ with unit the *diagonal* $\Delta = \{\pi_1 = \pi_2\}$; it is what makes the structure a dioid and what the *product* buys, since a bare set has one coordinate, nothing to compose and no diagonal to compare. The additive monoid is idempotent $(R \cup R = R)$, a join-semilattice; that half coincides with the Boolean-lattice join a relation carries *as a set* (Section 3, Table 2), which is why the two units differ: $\varnothing$ is the semilattice bottom, Δ the composition one, and the lattice top $A \times B$ is neither. Δ is graph(id), definable on every carrier because equality is universal. `ext` sends this dioid into the Boolean matrix semiring $\mathrm{Mat}(\mathbb{B})$ as a homomorphism, $\varnothing \mapsto 0$, $\Delta \mapsto I$, $\cup \mapsto \oplus$, $; \mapsto \otimes$; that instance is reachability.

$\exists^{\leq 1}$, distinct from the fibre's indexed meet $\bigwedge$ (which would instead demand relation to *every* element). A projection becoming an isomorphism needs both bounds at once: π_A an iso makes R a function, π_B an iso makes the *converse* a function, and both together a bijection. The epic-monic factorization $R = m \circ e$ places the retract on the mono leg m, where it later discharges image-membership and function application (§4.0.1).

	$\pi_A : R \to A$		$\pi_B : R \to B$	
	entire	functional	surjective	injective
Trsk (point-free)	$I \subseteq R \cdot R^{-1}$	$R^{-1} \cdot R \subseteq I$	$I \subseteq R^{-1} \cdot R$	$R \cdot R^{-1} \subseteq I$
set theory	total	single-valued	onto	one-to-one
category theory	π_A epi	π_A mono	π_B epi	π_B mono

Table 4: The four properties of a relation $R \subseteq A \times B$ as an existence bound (≥ 1) and a uniqueness bound (≤ 1) on its two projections, in three vocabularies. The top row is Tarski's point-free calculus of relations [37], with no bound variables: existence bounds are $I \subseteq (\cdot)$, uniqueness bounds $(\cdot) \subseteq I$, and the relative product $R \cdot R^{-1}$ is the DSL's `converse` and `>>`. Columns pair up: entire with functional on π_A, surjective with injective on π_B.

A relation that is both total and functional is a *function*. In the library this is the concept refinement `IsFunction = IsRelation \&\& IsEntire \&\& IsFunctional`, so `IsFunction` subsumes `IsRelation`: the inclusion {functions} $\subset$ {relations} is genuine subtyping in the concept lattice, not an analogy. The bridge from an analytic arrow to its relational form is `graph(f)`, the graph $\Gamma_f = \{(a, f(a)) \mid a \in A\}$; its `is_graph_of` witness makes "the rule and its set of pairs are the same function" a compile-time fact rather than a claim. These two readings, the intensional rule `f` and the extensional set of pairs Γ_f, are the horizontal axis of the same picture whose vertical axis is the relation/function refinement above, bridged by `graph` one way and enumeration (`ext`) the other.

Listing 7 is the spec surface: `IsRelation` as an `IsSet` on the product, `IsFunction` as its entire and functional refinement, and the calculus of relations whose composition `>>` functions

inherit as a special case.

```
template <class R,class A,class B>
concept IsRelation =
  IsSet<R> && IsProduct<Dom<R>,A,B>;

static_assert(IsPredicate<
  Comprehension<𝔸<pair<A,B>,Classical
  ↪  >,P>>);

template <class R> concept
IsEntire = is_left_total_v<R>;
template <class R> concept
IsFunctional = is_right_unique_v<R>;

template <class R,class A,class B>
concept IsFunction =
  IsRelation<R,A,B> &&
  IsEntire<R> && IsFunctional<R>;
```

```
template <class A,class B,class
↪  L,class P>
auto converse(Comprehension<𝔸<pair<A,
↪  B>,L>,P>);

template <class R,class S>
constexpr auto operator>>(R,S);

template <class R> concept
IsSurjective = is_right_total_v<R>;
template <class R> concept
IsInjective = is_left_unique_v<R>;
```

```
// inferred, not declared: leaf AND composite are functions BY STRUCTURE:
static_assert(IsFunctional<decltype(ℤ*ℤ & π_1+3==π_2)>);  // leaf: x+3
static_assert(IsFunctional<decltype(                          // composite:
  (ℤ*ℤ & π_1+3==π_2) >> (ℤ*ℤ & π_1+2==π_2))>);       //  inherited via >>
```

Listing 7: The spec surface, and where pluggability goes. **Left**: the core concepts, a relation as an `IsSet` on the product $A \times B$, a function as its entire, functional refinement. **Right**: the combinators of the calculus of relations, composition the same `>>` functions use. **Bottom**: the four properties are *pluggable* traits (trust at the leaves), but inferred where structure decides: a leaf carries one by its predicate shape, and a composite inherits it through `>>` (the compositional closure of Table 4's containments), so a graph and its relative product are functions *by structure*, with no opt-in flag.

```
constexpr auto R = ℕ * ℕ & (π_1 % 3 == π_2);
static_assert( R(pair{7,1}));
static_assert(!R(pair{7,2}));
```

Listing 8: The residue relation $\{(a, b) \in \mathbb{N} \times \mathbb{N} \mid b = a \bmod 3\}$ as a strongly-typed point-free predicate, and its pointwise validation. The divisor is a constant, so the predicate is total.

The design of the relation R in Listing 8 highlights the core architectural boundary between symbolic specification and type-level evaluation. Structurally, the relation is defined entirely through an intensional presentation. Rather than an opaque lambda over the pair, its predicate is built point-free from the projections π_1, π_2: a strongly-typed, type-level composition the compiler can inspect, not a closure it must treat as a black box. The modulus divides by the constant 3, whose nonzero value is checked when the term is built, so the predicate is total: a term divisor, which could be 0, is not admitted.

This intensional rule collapses into an extensional, pointwise evaluation context when interrogated by a concrete element via the indicator property `operator()`. During compilation, the `static_assert` forces the template engine to substitute the candidate literal pair into the predicate, reducing the symbolic constraint down to a primitive boolean literal. Consequently, the library maintains the abstract economy of mathematical logic right up until specific domain elements witness the correctness of the constraint at compile time.

In classical relational calculus, the relative composition of two relations $R \subseteq A \times B$ and $S \subseteq B \times C$ requires an existential quantification step over the intermediate domain B:

$$R \circ S = \{(a, c) \in A \times C \mid \exists b \in B : (a, b) \in R \wedge (b, c) \in S\} \tag{12}$$

On a finite (`IsExtensional`) intermediate domain B, that $\exists b$ is exactly the bounded existential of §3.3: relative composition is quantification over the middle set. General first-order existential ($\exists$) and universal ($\forall$) quantification cannot, however, be implemented generically over *infinite* intensional sets at compile time, as the enumeration of open predicate domains induces infinite compiler loop regression.

Over a *finite* intermediate this $\exists b$ enumerates and folds at compile time. Listing 9 composes $\leq$ with itself over $\mathbb{B}$: the relative product $R \gg S$, spelled with the overloaded `>>`, decides the $\exists b$ on the two Boolean witnesses and recovers $\leq$ (transitivity, $\leq \circ \leq = \leq$).

```
constexpr auto imp = 𝔹 * 𝔹 & π_1 <= π_2;
static_assert( (imp >> imp)(pair{false,true}));
static_assert(!(imp >> imp)(pair{true,false}));
```

Listing 9: Composition of relations as the relative product $R \gg S = \{(a, c) \mid \exists b.\, R(a, b) \wedge S(b, c)\}$. Over $\mathbb{B}$ the $\exists b$ enumerates the two Booleans and folds, so $\leq \circ \leq = \leq$ (transitivity).

Over an *infinite* intermediate the enumeration does not terminate: the $\exists b$ is then decidable only through one of the certificates of §3.3, the finite quotient or (on the function side) the retract of §4.0.1. The standard lazy alternative is the *monadic* presentation of a relation, $R : A \to \mathcal{P}(B)$, under which composition is Kleisli arrow composition over the power-set monad, $(R \gg S)(a) = \bigcup_{b \in R(a)} S(b)$, a symbolic pipeline that never materialises the middle set; it is the currying transpose of the relative product, so both are one relation composed two ways, differing only in which intermediate you can afford to enumerate.

Listing 10 spells relations out in the point-free surface: a comprehension over the product $A \times B$ whose predicate compares the projections π_1, π_2, with `converse` the one relational operator shown (functions follow in a later slice).

```
constexpr auto imp =
  𝔹 * 𝔹 & π_1 <= π_2;
static_assert(is_relation(imp));
static_assert( imp(pair{false,true}));
static_assert(!imp(pair{true,false}));
constexpr auto rimp = converse(imp);
static_assert(rimp(pair{true,false}));
```

```
static_assert(
  (ℕ * ℕ & π_1 < π_2)
  (pair{2,5}));
static_assert(
  (ℕ * ℕ & π_1 + 1 == π_2)
  (pair{4,5}));
static_assert(
  (ℕ * ℕ & π_1 * 2 == π_2)
  (pair{3,6}));
```

Listing 10: Relations as comprehensions over a product, point-free in the projections π_1, π_2. **Left** ($\mathbb{B}$): implication $a \leq b$ on $\mathbb{B} \times \mathbb{B}$, and its `converse`. **Right** ($\mathbb{N}$): less-than, and two *functional* graphs written point-free via the projection arithmetic: the successor $b = a{+}1$ and the doubler $b = 2a$.

The product itself is a first-class set: $A \times B$ is the universal set of pairs $\mathbb{A}_{A \times B}$. A *restricted* factor is then a refinement of that universe, not a special case, since $A \times B = \pi_1^{-1}(A) \cap \pi_2^{-1}(B)$. Restricting a factor lifts its predicate onto the matching projection, the same π_I vocabulary the graph uses, so the bound is carried rather than dropped (Listing 11).

The order playbook of §3.3 now meets a function. Before optimising over one, we settle what a function *is* on this surface, and how its image is decided.

```
constexpr auto R =
  (ℕ & (π <= 5)) * ℕ
  & π_1 + 1 == π_2;
static_assert( R(pair{4,5}));
static_assert(!R(pair{7,8}));
```

Listing 11: A restricted domain is a cylinder on the product. The bound ≤ 5 on the first factor lifts to $\pi_1 \leq \mathrm{fix}(5)$, so it survives into the relation instead of being discarded.

4.0.1 Functions and Morphism Factorization Gates

Image membership, $y \in \mathrm{image}(f)$, is an existential on the domain side: $\exists x.\, f(x) = y$. Over a finite or finite-quotient domain the certificates of §3.3 discharge it; over an unconstrained, transfinite domain neither applies, and a raw search of the domain is undecidable. The remaining handle is a *closed-form* certificate: a *retract*, the partial inverse of the mono leg of the map's epic-monic factorization, which *computes* the witness x rather than searching for it. This is not a rival to the quantifier but the case where the existential has a closed-form solution: the same $\exists x.\, f(x) = y$, discharged by construction. The library reifies it as an architectural gate based on the categorical principle of epic-monic factorization.

Mathematically, a relation $R : A \to \mathcal{P}(B)$ is a total function if and only if it maps every input to a singleton; equivalently, it is *functional* (single-valued) and *entire* (total). In the codebase this is the concept `IsFunction`: a relation refined by the right-unique and left-total witnesses, with `is_single_valued_at` discharging single-valuedness pointwise. A function is thus not a distinct primitive but a relation satisfying both properties, so `IsFunction` subsumes `IsRelation`: the inclusion {functions} $\subset$ {relations} is made mechanical by concept refinement.

Reifying functions in this manner allows the type system to mechanically verify classical mapping properties at translation time. For specialized mappings, codomain membership tests, checking whether a given element $b \in B$ lies within the actual image of the transformation, are easily discharged by the compiler. If a function is certified as surjective, every element of the target space is a valid image, meaning a codomain membership query trivially constant-folds to logical truth. More generally, if the mono leg of the factorization carries a *retract* (an inverse on the image, total when the arrow is an iso and partial when it is merely monic), the type system decides membership as a local lookup on the image boundary, with no search of the domain (Listing 12).

The four maps that read a relation, its `dom` and `cod` and the `image` and `preimage` of a set, are one cell each of Table 4: one bound on one projection. `preimage` and `dom` read the π_A side and never leave the domain, so they cost nothing beyond a forward call: $\mathrm{preimage}(f, S) = \{a \mid f(a) \in S\}$ is apply-and-test, decided by single-valuedness alone, and dom returns the *declared* first factor $\mathbb{A}\langle A\rangle$ (the effective domain $\{a \mid \exists b.\, R(a, b)\}$, which equals A only when f is entire, is the separate $\exists$-projection behind the Rice wall). `image` and `cod` read the π_B side: cod likewise returns the *declared* second factor $\mathbb{A}\langle B\rangle$ (all of B effectively only when f is surjective), and $b \in \mathrm{image}(f, S)$ is the question $\exists x \in S.\, f(x) = b$, which injectivity turns into the single lookup $f^{-1}(b) \in S$ and does have to cross back. The forward image is thus the one whose membership needs the inverse, and that inverse is the retract on the mono leg. An isomorphism meets all four bounds and collapses every cell to a constant-time answer; a plain function gets its π_A side for free and waits on injectivity for its π_B side, which is exactly where the retract is required.

Table 4 classifies the arrow; the same projection bounds fix the shape of its *inverse*, and with it the whole of image-membership. Every case is the one law

$$y \in \mathrm{image}(f, S) \iff \bigvee_{x \in f^{-1}(y)} S(x), \tag{13}$$

the fibre $f^{-1}(y)$ met against the source S; what changes across Table 5 is only the fibre's *shape*. An iso or a mono has a fibre of at most one point, so the join is a single lookup through a total or partial inverse. An epi or a general map has a many-point fibre, which is a point-free *union* of branches, so the join is the union $\bigcup_i \text{preimage}(b_i, S)$ of the branches' pullbacks, decided analytically with no enumeration. The one-sided inverses all compose *contravariantly*, so retractability and section-ability are closed under composition and, in particular, are preserved by pre- or post-composition with an iso (Listing 12).

We note a deep connection between converse and inverse, which Table 5's final column summarises. Each factorisation class has a *dagger refinement* in which the dagger itself supplies the one-sided inverse: an isometry ($f^\dagger f = \text{id}$) is a split mono with retract $f^\dagger$, a coisometry ($ff^\dagger = \text{id}$) a split epi with section $f^\dagger$, a unitary (both) an iso with $f^{-1} = f^\dagger$, and a partial isometry ($ff^\dagger f = f$) a general arrow factoring as a dagger-mono after a dagger-epi. The implication runs one way (an ordinary mono need not be an isometry), but where it holds "the converse is the inverse" (a bijective relation) and "the transpose is the inverse" (an orthogonal matrix) are one theorem. The interested reader is referred to [38, 39, 40, 41].

arrow f	one-sided inverse	fibre $f^{-1}(y)$	composition $h = g \circ f$	†-refinement
iso	f^{-1} (two-sided)	$\{f^{-1}(y)\}$	$h^{-1} = f^{-1} \circ g^{-1}$	unitary
mono / injective	retract r, $r \circ f = \text{id}$	$\{r(y)\}$ if $f(r(y))=y$, else $\varnothing$	$r_h = r_f \circ r_g$	isometry
epi / surjective	section s, $f \circ s = \text{id}$	a union, containing $s(y)$	$s_h = s_f \circ s_g$	coisometry
general $m \circ e$	g, $fgf = f$	$\bigcup_i \{s_i(y)\}$	converse composes contravariantly	partial isometry

Table 5: How Table 4's arrow class fixes the inverse structure, hence image-membership (Equation 13) and its composition law. The fibre is a single point (iso, mono) or a point-free union (epi, general); image is the fibre met against the source, and the one-sided inverses all compose contravariantly, so retractability is closed under composition. The final column names each class's *dagger refinement*: the special case, in a dagger category, where the dagger itself supplies the one-sided inverse (unitary $f^{-1} = f^\dagger$, isometry retract $r = f^\dagger$, coisometry section $s = f^\dagger$, and for a general $m \circ e$ that factors as a dagger-mono after a dagger-epi, partial isometry with $g = f^\dagger$, the Moore–Penrose pseudoinverse in the Hilbert case). The refinement runs one way (an ordinary mono need not be an isometry), and partial isometries, unlike the other three classes, are not closed under composition.

The graph is the whole surface: the same $*$, $|$, and projection arithmetic that spell a relation spell a function, and `inverse` and `image` are relational operations on it, not a separate arrow calculus. Because a translation is surjective its image is decided by the inverse alone, so a bounded halfspace pushes forward to a halfspace with its bound advanced while the unbounded line is fixed. The non-injective case drops the injectivity and stays point-free. Take the sign-fold $\text{abs}\colon \mathbb{Z} \twoheadrightarrow \mathbb{N}$, the epi that identifies x with $-x$. It is two-to-one, the fibre of y being $\{+y, -y\}$, so it is *not* monic; but it is a *union* of two reflection branches, $x \mapsto x$ on $\{x \geq 0\}$ and $x \mapsto -x$ on $\{x < 0\}$, each a mono whose image is a halfspace pushed forward (Listing 13). Membership $y \in \text{abs}(S)$ is the join of the branch images, Equation 13, decided analytically with no fibre walk. That join is what keeps it sound: $3 \in \text{abs}(\{x < 0\})$ holds through the negate branch, which carries the preimage -3, where the canonical $+3 \notin \{x < 0\}$ that a lone retract would name says nothing. This is also what discharges the arg max of §4 over an infinite domain: an optimiser and its fibre are the same $\exists x.\, f(x){=}y$, read off the union of branches rather than searched for.

The factorization here is *consumed* point-free, not narrated: `image` is the join of the two branch images (Equation 13), so the non-injective case is decided analytically rather than walked. Not every map is analytic like abs, though: for a general function whose fibre is *not* a union of graph branches, validating whether an arbitrary element $b \in B$ possesses a

```
constexpr auto f =                          //
↪  x ↦ x+3
  ℤ * ℤ & π_1 + 3 == π_2;
static_assert(is_function(f));
static_assert(inverse(f)                    //
↪  converse: x-3
  == (ℤ*ℤ & π_1+-3==π_2));
static_assert(image(f|π_1<=5) //
↪  {x≤5}↦{y≤8}
  == (ℤ & π<=8));
```

```
constexpr auto g =                          //
↪  x ↦ x+2
  ℤ * ℤ & π_1 + 2 == π_2;
static_assert((f >> g)                      //
↪  closure: T₃∘T₂=T₅
  == (ℤ*ℤ & π_1+5==π_2));
static_assert(inverse(f >> g)               //
↪  contravariant: (f∘g)⁻¹=g⁻¹∘f⁻¹
  == (inverse(g) >> inverse(f)));
static_assert(                              //
↪  g∘f∘g⁻¹ = f (abelian)
  (g >> f >> inverse(g)) == f);
```

Listing 12: A function *is* its graph, and translations form a *group* under composition. **Left**: `f = ℤ*ℤ & π_1+3==π_2` is the point-free relation $\{(x, x{+}3)\}$, functional and entire (Table 4's π_A column); its `inverse` is the converse relation $x{-}3$, and `image` pushes a domain bound forward, $\{x \le 5\}$ to $\{y \le 8\}$, again a halfspace. **Right**: composition `>>` is symbolic (the shifts add, no $\exists$ over the intermediate), so closure, the inverse-of-a-composite law $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$, and *associativity, commutativity* are compile-time facts; the conjugate $g\,f\,g^{-1} = f$ collapses because $+$ commutes. The law is contravariant in general, but this group is abelian, so the order-reversal is degenerate here ($g^{-1} \circ f^{-1} = f^{-1} \circ g^{-1}$); the non-abelian affine group exercises it. For an invertible map the inverse *is* the retract, so this is retract composition in the total case (abridged from the checked-in witnesses).

```
constexpr auto absPos = ℤ*ℤ & π_1*1==π_2  & π_1>=0;  // x ↦  x  on x≥0
constexpr auto absNeg = ℤ*ℤ & π_1*-1==π_2 & π_1< 0;  // x ↦ -x  on x<0
static_assert(image(absPos) == (ℤ & π>=0));  // {y≥0}
static_assert(image(absNeg) == (ℤ & π> 0));  // {y>0}: negate flips the sense
static_assert( image(absNeg)(3));   // 3 ∈ abs({x<0}) via -3, no fibre walk
static_assert(!image(absNeg)(-2));  // abs is never negative
```

Listing 13: The sign-fold epi abs: $\mathbb{Z} \twoheadrightarrow \mathbb{N}$ decided *point-free*: a union of two reflection branches ($x \mapsto x$ on $\{x \ge 0\}$, $x \mapsto -x$ on $\{x < 0\}$), each a mono whose `image` is a halfspace pushed forward (the negate branch flips $\le$ to $>$). Non-injective yet analytic: $3 \in \mathrm{abs}(\{x < 0\})$ holds through the negate branch's preimage -3, where the canonical $+3 \notin \{x < 0\}$ that a lone retract names says nothing: soundness from the fibre join, no enumeration (checked-in witnesses).

valid preimage in the domain A presents a severe decidability barrier if evaluated via standard search techniques. Rather than treating this as an impassable restriction, the library provides specialized support for functional relations by introducing an architectural gate based on the categorical principle of epic-monic factorization. Every general functional arrow $f : A \to B$ factors into a regular surjection (epi, `IsRegularEpi`) onto its image I, followed by a regular injection (mono, `IsRegularMono`) from I into B, the image leg witnessed by `IsImageOf`:

$$A \twoheadrightarrow I \hookrightarrow B. \tag{14}$$

By requiring the developer to provide this factorization when declaring an arbitrary `IsFunct⌋ion` (equivalently, an `IsArrow` whose graph is functional), the compiler gains the exact structural properties needed to handle general codomain membership tests. Since the second leg of the factorization chain is a strict injection, it guarantees a one-to-one mapping from the intermediate image set I into the target space B, allowing the type-checker to reason about the embedding without exploring the raw domain space A. This gate *relocates* the guarantee rather than deriving it: the factorization is a *trusted axiom*, the C++ analogue of Coq's `Admitted`, checked for shape (`IsRegularEpi`, `IsRegularMono`, `IsImageOf`) but not for lawfulness. A factorization that is registered but wrong is unsound: the compiler trusts it, so a downstream membership test can fold a false constant into the emitted IR with no diagnostic. This is the honesty obligation of §1 at the level of morphisms, and it marks where a standard C++ template engine reaches an undecidability wall and asks the developer to supply the witness it cannot itself compute. Listing 14 illustrates how this epic-monic gate and its corresponding image validation assertions are codified within the C++23 template engine.

```
constexpr auto succ =          // as a
↪  graph
  ℕ * ℕ & π_1 + 1 == π_2;
static_assert(
  image(succ & π_1 > 5)  // {n>5}
  == (ℕ & π > 6));       //  ↦ {n>6}
```

```
constexpr auto op =            // opaque
↪  λ
  arrow<Cardinality,Cardinality>(
    [](Cardinality m){ return m + 1;
    ↪  });
constexpr auto img =
  image(op, ℕ & (π > 5));
static_assert(                 // honest
↪  wall
  img(finite_cardinality(7)) ==
  ↪  Ternary::Unknown);
```

Listing 14: Structure decides where opacity walls. **Left**: the successor as a *graph*; its image over $\{n > 5\}$ is the affine pushforward $\{n > 6\}$, decided with no search of the domain. **Right**: the same map as an *opaque* arrow the compiler cannot see through, so the existential $\exists x.\, x > 5 \wedge y = \mathrm{succ}(x)$ is undecidable on the transfinite carrier: `image` realises `IsImageOf` yet its membership is honestly `Unknown`. The Rice boundary is for opacity, not for functions (abridged from the checked-in witnesses).

By utilizing the `retract` partial witness of an `IsRetractableArrow` (the mono leg's partial inverse, returning `std::optional`), the compiler evaluates codomain membership as a local lookup on the image boundary rather than a search of the domain. This structural approach ensures that even highly complex or non-linear functional images can be safely validated at compile time. The type system can consequently treat arbitrary transformations as well-behaved objects, the foundation on which the optimisation that closes this section, and the universal algebra beyond it, are built.

4.0.2 Optimization: arg max over a Function

With functions and their images in hand, optimising over one is the same order playbook read through a map. Reading $\leq$ through $f\colon \mathrm{dom} \to C$ turns max into arg max: not the greatest *value* but every *argument* that attains it. The definition is the forward filter of Equation 9, with the carrier's own order replaced by the order f pulls back onto its domain, and it asks nothing of f beyond a comparable codomain: no inverse, no injectivity.

$$\begin{aligned} \operatorname*{arg\,max}_{f} &:\equiv \{x \in \mathrm{dom} \mid \forall x' \in \mathrm{dom}.\ f(x') \leq f(x)\}, \\ \arg\min &:\equiv \arg\max \text{ under } \leq \mapsto \geq. \end{aligned} \tag{15}$$

Read as a relation this is the pulled-back comparison $f(\pi_1) \leq f(\pi_2)$ on $\mathrm{dom} \times \mathrm{dom}$, universally quantified over π_1: Listing 10's $\pi_1 \leq \pi_2$ with each side seen through f. Because the answer is a set, the degenerate cases fall out as a *cardinality* rather than as special code: the winning fibre is empty when there is no maximum, a singleton exactly when the optimiser is unique (so arg max is a function), and larger under ties (so arg max is a proper relation), the same fibre bookkeeping Table 4 reads off a projection. Constraining the codomain, as in §4.0.1, turns the graph into a partial function whose feasible domain is the codomain constraint pulled back through it; a monotone objective then makes arg max that domain's greatest element (Listing 15).

```
constexpr auto g = ℤ * ℤ & π_1 + 3 == π_2   // x ↦ x+3 ...
                   & (π_2 <= 8               //   ... into a bounded ...
                     && π_2 % 3 == 0); // ... residue codomain
static_assert(!is_entire(g));                 // a partial function now
static_assert( argmax(g)(3));                 // max{x≤5, x≡0 mod 3} = 3
static_assert(!argmax(g)(4));                  // 4 ≢ 0 mod 3
```

Listing 15: arg max over a *partial function*: a compile-time constrained optimum. Constraining the codomain of the graph $x \mapsto x + 3$, bounded ($\pi_2 \leq 8$) and restricted to a residue class ($\pi_2 \equiv 0 \bmod 3$), drops *entire* but keeps *functional* (Table 4): the same graph is now a partial function. Its feasible domain is that codomain constraint pulled back through the graph, $\{x \leq 5 \wedge x \equiv 0 \bmod 3\}$, and because a translation is monotone arg max is that domain's greatest element, 3, read off structurally with no search.

The optimum is not searched for but read off the structure: the codomain bound and the congruence, pulled back through the graph, fix the feasible domain, and a monotone objective makes arg max its greatest element. The implemented arg max is exactly this structural specialization (a group-carrier translation into a codomain bounded above and restricted to a residue class); a general filter over arbitrary, less-structured objectives is future work, decidable only where §3.3's certificates reach.

5 A Reification of Alg(−)

This section is the Galois specialisation (Definition 2) of **Alg**(−), the category of varieties. **Alg**(−) is reified as the *relational magmas* over **Trsk**: a carrier with a pointwise operation $A \times A \to A$, read as the functional relation that is its graph and refined by the laws it keeps (Equation 17). The section is four capabilities, each a compile-time certificate (Listing 16): a magma is a carrier with a closed operation, and Birkhoff's three constructions — quotient (H), subalgebra (S), and direct product (P) — each build a new carrier from it, the base's certificate carried across for S and P by trait propagation, and, for the relational H shown here, the quotient *witnessed* by its congruence rather than by propagation (§4).

magma — a carrier with a closed operation

```
// ℤ/8ℤ: + closes on the carrier
static_assert(IsMagma<Modular<8u>,
                      std::plus<Modular<8u>
                      ↪  >>);
```

S — subalgebra (a closed subset)

```
// even ⊆ (unsigned, +): total ambient
static_assert(IsSubalgebra<EvenUnsign
↪  ed,
    unsigned, std::plus<unsigned>>);
```

H — quotient by a congruence

```
// Modular<8> = unsigned/(≡ mod 8),
↪  8|2^w
static_assert(IsCongruenceQuotient<
    Modular<8u>,
    ↪  std::plus<unsigned>>);
```

P — direct product (componentwise)

```
// Vec2<T> = T × T
static_assert(
    IsProductAlgebra<Vec2V<int>>);
```

Listing 16: Section 5 at a glance: four capabilities, each a checked-in `static_assert`. A magma is a carrier with a closed operation; the three Birkhoff constructions each build a new carrier from it — S and P carry the certificate across by trait propagation, while the relational H *witnesses* the quotient by its congruence. The section develops each below.

Two facts organise the columns. Every homomorphism $f\colon A \to B$ factors through its image, an epimorphism π (a quotient, H) followed by a monomorphism ι (a subalgebra inclusion, S), the arrow classes of Table 4:

$$A \twoheadrightarrow \operatorname{im}(f) \hookrightarrow B, \qquad f = \iota \circ \pi. \tag{16}$$

And because **Jlt** materialises only finite carriers (the model/materialise gap of §3, Equation 10), of the three constructions only the product carries an arity bound.

Table 6: The three Birkhoff constructions H, S, P read across vocabularies; the arrow-class row is Equation 16, the finitary bound Equation 10.

	H (quotient)	**S** (subalgebra)	**P** (product)
universal algebra	A/θ, a congruence θ	closed $B \subseteq A$	direct product $\prod_i A_i$
arrow class	epi (surjection)	mono (injection = embedding)	product (limit)
relation (**Trsk**)	a congruence relation	a closed predicate	componentwise on a tuple
concept	`IsCongruenceQuotient`	`IsSubalgebra`	`IsProductAlgebra`
exemplar	$\mathbb{Z}/N\mathbb{Z}$, $F[\varepsilon]/(\varepsilon^2)$	$\mathbb{N} \subseteq \mathbb{Z}$	T^n (vector, matrix)

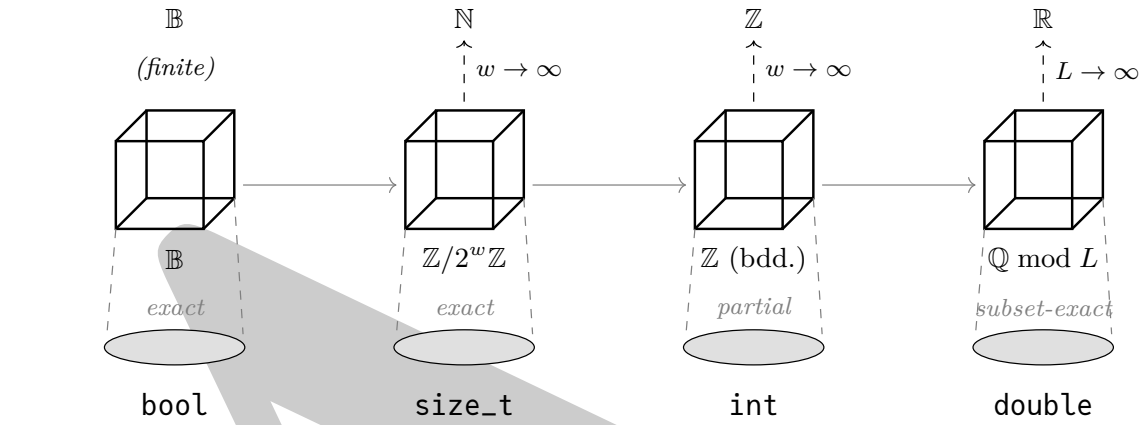
5.1 Algebraic Sets on the Primitive Carriers

The primitive machine carriers sit at the base of the tower as shadows of finite modular Forms (Figure 3): `bool` ($\mathbb{B}$) and `size_t` ($\mathbb{Z}/2^w\mathbb{Z}$) exactly, `int` as a *partial* $\mathbb{Z}$ (overflow is undefined), and `double` exactly on its integer / bounded-dyadic subset (rounding the price of leaving it); the infinite ideals ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$) are the period-∞ limits.

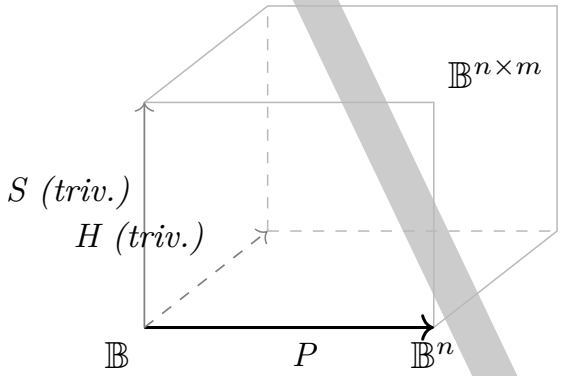
Listing 17 reifies the *algebraic sets* of Figure 1 as C++23 types: the family **Ddk** is the point-free relational calculus **Trsk** (11) met with universal algebra in the textbook sense [42], **Alg**$(-)$[9]:

$$\mathbf{Ddk} \;=\; \mathbf{Trsk} \cap \mathbf{Alg}(-). \tag{17}$$

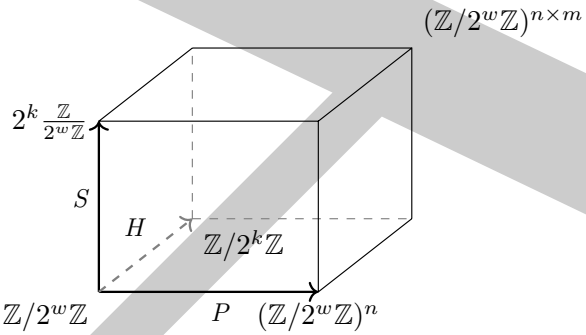
[9]The dash is left unbound, the same F_Σ-algebra fragment a functional programmer would recognise [32]. The smallness ceiling that keeps **Ddk** clear of proper-class self-reference is the library's Axiom-9 restriction of carrier predicates to $\beth_1$ (Appendix C), not a general ETCS bound.



(a) the shadow plot: four primitives as shadows of modular Forms



(b) HSP on `bool`: P-rich, S/H trivial — $\mathbb{B}^n$, $\mathbb{B}^{n\times m}$ (componentwise)



(c) HSP on `size_t`: full H/S/P; H reduces $\mathbb{Z}/2^k\mathbb{Z} \to \mathbb{B}$

Figure 3: The primitive carriers, two readings stacked. **(a)** The *shadow plot* (vertical axis): each machine type is a shadow of a finite modular Form — *exact* where the Form is finite (`bool`= $\mathbb{B}$, `size_t`= $\mathbb{Z}/2^w\mathbb{Z}$), and exact only on a *closed subset* otherwise. `int` is a *partial* $\mathbb{Z}$ (signed overflow is undefined); `double` is exact on its integer / bounded-dyadic subset — closed and associative there ($1 + 1 + 2$ is exact), while general + rounds, the price of leaving it — approximating $\mathbb{Q} \bmod L$. The ideals $(\mathbb{N}, \mathbb{Z}, \mathbb{R})$ are *approached*, not lost: as the period grows ($w, L \to \infty$) the Form becomes an ever more faithful shadow, and $\mathbb{R}$ is the *completion* no finite stage reaches (Equation 10). The horizontal tower is *not* one construction; that lateral structure lives in the cubes, not a single arrow. **(b,c)** Two Forms opened as HSP-varieties (Figure 1's constructions, re-based). On `bool`, S and H are trivial ($\mathbb{B}$ is simple), so P carries it: $\mathbb{B}^n$ and the *componentwise* $\mathbb{B}^{n\times m}$ (all finite Boolean algebras, by Stone). On `size_t` all three legs are live over the *additive group* $(\mathbb{Z}/2^w\mathbb{Z}, +)$ — P the componentwise arrays, S the subgroups $2^k\mathbb{Z}/2^w\mathbb{Z}$, H the quotient groups $\mathbb{Z}/2^k\mathbb{Z}$ (at $k = 1$, $\mathbb{B}$, so the `bool` cube *is* the H-corner); the multiplicative structure would leave S trivial (1 generates the whole ring). Birkhoff P is *componentwise*; the relational / matrix product $M_n(\mathbb{B}) = \mathbf{Rel}$ (summing over an index) is the separate *endomorphism* construction — composition, §4 (Table 7) — not P. HSP furnishes the cubes; matrix multiplication and the vertical limits are the axes it does not span.

type layer (`category`): pointwise

```
template <typename T, typename Op>
concept IsMagma  = IsTotal<T, Op>
                && IsClosedUnder<T,
                 ↪  Op>;
template <typename T, typename Op>
concept IsMonoid = IsMagma<T, Op>
   && IsAssociative<T, Op>
   && IsPointed<T, Op>;
```

lift base (`algebra`): `IsAlgebraOnSet`

```
template <typename X, typename... Ops>
concept IsAlgebraOnSet = IsSet<X>
  && IsClosedAlgebra<typename
  ↪  X::Domain,
                     Ops...>;
```

the lifted rung (`algebra`): `IsMonoid`

```
template <typename X, typename Op = std::plus<typename X::Domain>>
concept IsMonoid = IsAlgebraOnSet<X, Op> && category::IsMonoid<typename
↪  X::Domain, Op>;
```

Listing 17: **Ddk = Trsk ∩ Alg**, in three panels. **Top-left, the type layer** (`category`): the pointwise axioms on a carrier *type*: a model of a Birkhoff/Lawvere theory [42, 43], a magma refined to a monoid by associativity and a unit. **Top-right, the lift base** (`algebra`): `IsAlgebraOnSet`, a set *object* whose carrier is closed: the subobject-over-carrier pattern familiar from proof assistants [44], i.e. **Trsk** met with carrier closure. **Bottom, a lifted rung**: `algebra::IsMonoid` is the lift base refined by the pointwise monoid axioms of the top-left.

𝔹 as an algebraic set (explicit ops)

```
// 𝔹 under logical (∨, ∧): a Boolean rig
static_assert(IsRig<decltype(𝔹),
    std::logical_or<bool>,
    std::logical_and<bool>>);
// 𝔹 as the Galois field 𝔽₂ (bitwise ⊕,
↪  ∧)
static_assert(IsGaloisField<decltype(𝔹),
    std::bit_xor<bool>,
    std::bit_and<bool>>);
```

𝕊 as an algebraic set (default ops)

```
constexpr auto 𝕊 =
↪  sets::𝔸<std::size_t>{};
// (+, ×) are the defaults, so
↪  omitted:
static_assert(
  IsCommutativeRing<decltype(𝕊)>);
```

Listing 18: Primitive carriers as *algebraic sets*, inhabiting the set-indexed concepts of Listing 17. **Left:** $\mathbb{B} = \mathbb{A}\langle\texttt{bool}\rangle$ needs *explicit* operators: a Boolean rig under logical $(\vee, \wedge)$ and the Galois field $\mathbb{F}_2$ under bitwise $(\oplus, \wedge)$. **Right:** $\mathbb{S} = \mathbb{A}\langle\texttt{size_t}\rangle$, the modular ring $\mathbb{Z}/2^w\mathbb{Z}$ (width w implementation-defined), is a commutative ring under the *default* arithmetic $(+, \times)$, so the operators are omitted. The complementary set-lattice layer $(\cup, \cap)$ is Listing 3; composite carriers such as $\mathbb{Q}$ appear in §5.2.

Listing 18 exhibits two primitive algebraic sets: $\mathbb{B}$ (a Boolean rig under $\vee, \wedge$, its carrier the field $\mathbb{F}_2$ under $\oplus, \wedge$) and $\mathbb{S}$ (the modular ring $\mathbb{Z}/2^w\mathbb{Z}$, width w implementation-defined, under the default $+, \times$); their set-lattice layer is Listing 3, and composite carriers such as $\mathbb{Q}$ follow in §5.2.[10]

Table 7: The universal constructions that build the carrier poset, the theorem by which each preserves algebraic laws, and whether the trait engine mechanizes it. The three Birkhoff operations [42] (product, quotient, subalgebra) share one closure theorem and *are* Rule 2; localization [45] and completion [46] preserve structure by their own theorems, from other branches, and are not Birkhoff-closed (fields are not a variety, so no set of identities pins them down).

construction	builds	law-preservation	in the engine
product / free module	scalar $\to$ vector	Birkhoff HSP (P) [42]	yes
quotient by an ideal	$\mathbb{R} \to \mathbb{C}, \mathbb{D}$	Birkhoff HSP (H) [45]	yes
subalgebra	a closed sub-part	Birkhoff HSP (S) [42]	mechanism only
localization (fractions)	$\mathbb{Z} \to \mathbb{Q}$	commutative algebra [45]	by hand
completion	$\mathbb{Q} \to \mathbb{R}$	analysis, Dedekind cuts [46]	by hand
endomorphism algebra	$R^n \to M_n(R)$	module theory [47]	by hand

5.2 Mechanical Support for Advanced Carriers

Beyond the primitives, mechanical support comes from Birkhoff's HSP theorem [42]: a certificate proven for a base carrier propagates to the carriers built from it. Two rules carry it (Listing 19):

1. **Composition on the parameter.** A concept conjoins a certificate with a property of its parameter: a module over a field *is* a vector space, so `is_vector_space_v` holds with no per-carrier code.

2. **Propagation through a construction.** A one-line registration of a base carrier — for a product, a subalgebra, or a quotient — forwards the species traits through the construction to the derived carrier. A quotient may *additionally* name its congruence relation (Listing 19), which *witnesses* $Q = V/R$; the law-forwarding itself rides the base-carrier registration, not the congruence.

This reaches exactly the Birkhoff-closed constructions, product, quotient, and subalgebra; localization, completion, and the endomorphism-algebra step preserve structure by other theorems and are not covered (Table 7).

5.3 Applicability to Linear Algebra

Linear algebra over these carriers takes two forms. Materialised, an operator is the matrix ring $M_n(F) = \mathrm{End}(V)$, the top of a ladder that type-checks over the rationals from scalar to matrix (Listing 20); the identity $M_n(F) = \mathrm{End}(V)$ is a registered declaration rather than a derived concept, since no concept recovers it from the operator surface alone (the honesty obligation of §1).

Coordinate-free, an operator never builds the grid: the coboundary δ, with $(\delta d)(u \to v) = d(v) - d(u)$, and its adjoint the boundary ∂ are expressions over the incidence structure, the intensional reading of §3 lifted from sets to operators. The practical rule is one line: constrain generic code on the concept, not the carrier; a routine bounded by `IsVectorSpace<V, F>` admits every carrier the closed core certifies, from $\mathbb{Q}$-vector-spaces to matrices over the dual numbers, and outside the core the certificate is the engineer's to supply.

[10] The two operator layers carry two *different* theories. The pointwise operations $(+, \times)$ make the carrier a model of a finitary Lawvere theory, i.e. the algebras of a free-Σ term monad [43]; the set operations $(\cup, \cap)$ come from the powerset functor, whose Eilenberg–Moore algebras are complete lattices, not rings. The set-indexed concept pairs the two, and is deliberately *not* the Eilenberg–Moore construction of the powerset monad.

the relational H combinator

```
// a quotient Q = V/R is witnessed by a
// CONGRUENCE relation R on the carrier
↪  V:
template <typename Q>
struct quotient_congruence {};
//   ::type = R,  ::carrier = V

template <typename Q, typename Op>
concept IsCongruenceQuotient =
  requires {
    typename
    ↪  quotient_congruence<Q>::type;
    typename
    ↪  quotient_congruence<Q>::carrier;
  } && IsCongruence<
    quotient_congruence_t<Q>,
    quotient_congruence_carrier_t<Q>, Op>
  // ...and Q reduces from V (V ↠ Q):
  && requires(
       quotient_congruence_carrier_t<Q>
       ↪  v)
     { { Q{v} } -> std::same_as<Q>; };
```

a success: clock arithmetic $\mathbb{Z}/N\mathbb{Z}$

```
// x ≡ y (mod N): the kernel of the
↪  reduction
// ℤ ↠ ℤ/Nℤ = Modular<N>, a genuine
// congruence for both + and ×.
template <auto N>
struct ModularCongruence {
  constexpr bool operator()(auto
  ↪  a, auto b)
  const { return Modular<N>{a}
               == Modular<N>{b}; }
};
template <auto N>
struct quotient_congruence<Modular
↪  <N>> {
  using type    =
  ↪  ModularCongruence<N>;
  using carrier =
    typename
    ↪  Modular<N>::machine_type;
};
// a congruence for native + iff
↪  N|2^w;
// Modular<8> = unsigned/(≡ mod 8):
static_assert(IsCongruenceQuotient<
    Modular<8u>,
    ↪  std::plus<unsigned>>);
```

Listing 19: The relational H (quotient) leg: a quotient $Q = V/R$ is declared by naming its *congruence* relation R on the carrier V (left), and `Modular<N>` $= \mathbb{Z}/N\mathbb{Z}$ names the mod-N relation as its congruence (right). `IsCongruence` checks that R is an equivalence preserved by the operation, so `IsCongruenceQuotient` holds — on the machine carrier $\mathbb{Z}/2^w\mathbb{Z}$ this needs N a power of two ($N = 8$ here), so that native wraparound agrees with mod-N reduction — and the certificate rides the relation. Carriers whose pre-quotient is not itself a reified carrier (the dual numbers $\mathbb{D} = F[\varepsilon]/(\varepsilon^2)$, $\mathbb{C}$) instead declare a trait-propagation base; the P (product) and S (subalgebra) legs mirror the same shape.

```
using Q = Rational<default_integer>;          // the ℚ carrier = Frac(ℤ)
constexpr auto ℚ = sets::𝔸<Q>{};              // ℚ as an algebraic set
using V = Vec2V<Q>;  using M = Matrix2x2V<Q>;

static_assert(IsField<decltype(ℚ)>);          // scalar: ℚ is a field
static_assert(is_module_v<V, Q>);             // module
static_assert(is_vector_space_v<V, Q>);       // vector space  (Rule 1)
static_assert(is_endomorphism_ring_v<M, V>); // operator: M₂ = End(V)
```

Listing 20: The certificate ladder over $\mathbb{Q}$: four rungs, each a `static_assert` the compiler discharges.

6 Discussion and Future Work

Two limitations of the present artefact point in the same direction. First, the real numbers are handled only rudimentarily. `dedekind`'s $\mathbb{R}$ is a Dedekind-cut carrier whose comparison is only

semi-decidable, and most numeric work still bottoms out in IEEE `double`, which is not cleanly algebraic: rounding defeats associativity and `NaN` punctures the order. Second, the symbolic algebra folds at compile time only over exact carriers such as $\mathbb{Q}$ and $\mathbb{Q}[\varepsilon]/(\varepsilon^2)$, not over the floating-point types production code runs on.

A natural way to reconcile the two is a *controlled boundary*, in the sense of staged, multi-stage programming [48]. Write the computation as a symbolic algebraic expression over an exact ring where the laws genuinely hold, let **Jlt** simplify and prune it at translation time, and only then lower the residual to IEEE floating-point for execution. The rewrites are sound because they happen on the exact ring. The lossy step is a single explicit evaluation map at the boundary, not a silent assumption threaded through every operation. Exactness earns a second dividend, on an axis orthogonal to that lowering. The lowering changes an expression's *representation*, from the exact ring to floating-point, and that map is lossy; changing only *when* the expression is evaluated, from translation time to runtime, is not. So the single lossy crossing is the representation map, and it is independent of where evaluation falls: an exact term yields the same value folded early or computed late. This is the ring-swap machinery the `Dual<F>` carrier already uses in miniature, generalised from computing a derivative exactly to computing the whole expression exactly and rounding once. Whether the type system can carry that boundary crossing as a first-class, mechanically-checked seam rather than a convention is the question we would pursue next.

A complementary direction widens the family of structural specialisations the compiler recognises (§3.3) toward a law-indexed reduction surface: gating the collapse rules on algebraic-law concepts such as `IsMonoid` and `IsMeetSemilattice` rather than on named carriers. Rice's theorem forecloses a fully general reducer. The family of law-carrying specialisations, however, has no obvious ceiling short of it.

References

[1] *<Programming>: The Art, Science, and Engineering of Programming*. `https://programming-journal.org/`. Open-access journal for programming-related research; ISSN 2473-7321.

[2] Vincent Kräutler. *dedekind*. Version v0.20261027.0+a1daef6. Apache-2.0. The listings of this paper are the tests `jlt_listings_test.cpp` and `tensor_exhibit_test.cpp`; the linter of Section 7 is `tools/jlt-lint`. 2026. DOI: `10.5281/zenodo.23697077`. URL: `https://github.com/vincentk/dedekind` (visited on 10/24/2026).

[3] Vincent Kräutler. "The Juliet Posture: Structurally Typed Functional Programming in Standard C++23". Draft, in the repository's `docs/paper/jlt`. 2026.

[4] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN: 9780262162098.

[5] William F. Lawvere. "An Elementary Theory of the Category of Sets". In: *Proceedings of the National Academy of Sciences* 52.6 (1964), pp. 1506–1511.

[6] ISO/IEC JTC 1/SC 22. *ISO/IEC 14882:2024: Programming languages — C++*. Commonly referred to as C++23. International Organization for Standardization. Geneva, Switzerland, 2024. URL: `https://www.iso.org/standard/83626.html`.

[7] Philip Wadler. "Propositions as types". In: *Communications of the ACM* 58.12 (2015), pp. 75–84. DOI: `10.1145/2699407`.

[8] Conal Elliott. "The simple essence of automatic differentiation". In: *Proceedings of the ACM on Programming Languages* 2.ICFP (2018), pp. 1–29.

[9] Wenzel Jakob. "Enoki: Structured Vectorization and Differentiation on Modern Processor Architectures". In: `https://github.com/mitsuba-renderer/enoki`. 2019.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, et al. “Evaluating Large Language Models Trained on Code”. In: *arXiv preprint arXiv:2107.03374* (2021). DOI: 10.48550/arXiv.2107.03374.

[11] Robert David et al. “TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems”. In: *Proceedings of Machine Learning and Systems (MLSys)*. 2021.

[12] Lennart Augustsson. *Djinn: Generate Haskell Code from a Type*. Haskell package. Synthesises an inhabiting term from a type via intuitionistic propositional proof search. 2005. URL: https://hackage.haskell.org/package/djinn (visited on 09/15/2026).

[13] Roy Dyckhoff. “Contraction-Free Sequent Calculi for Intuitionistic Logic”. In: *The Journal of Symbolic Logic* 57.3 (1992). The terminating LJT calculus underlying Djinn’s decision procedure., pp. 795–807. DOI: 10.2307/2275431.

[14] Henry Gordon Rice. “Classes of Recursively Enumerable Sets and Their Decision Problems”. In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. DOI: 10.1090/S0002-9947-1953-0053041-6.

[15] Roger Penrose. *The Road to Reality: A Complete Guide to the Laws of the Universe*. Chapter 16 and surrounding material on Gödel’s incompleteness theorem. London: Jonathan Cape, 2004.

[16] David Geiger. “Closed Systems of Functions and Predicates”. In: *Pacific Journal of Mathematics* 27.1 (1968). The Pol–Inv Galois connection: a dual isomorphism between clones of operations and coclones of the relations they preserve., pp. 95–100.

[17] Dmitriy Zhuk. “A Proof of the CSP Dichotomy Conjecture”. In: *Journal of the ACM* 67.5 (2020). The complexity of a fixed-language CSP is determined by its polymorphisms (operation-preserving invariants): tractable given a weak near-unanimity polymorphism, NP-complete otherwise., pp. 1–78.

[18] Emmy Noether. “Invariante Variationsprobleme”. In: *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse* (1918). Noether’s theorem: continuous symmetries are coupled to conserved invariants., pp. 235–257.

[19] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Classification of the decidable fragments (guarded, two-variable, monadic, Bernays–Schönfinkel) of an undecidable first-order theory. Springer, 2001.

[20] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[21] Stephen Cole Kleene. “On the Interpretation of Intuitionistic Number Theory”. In: *The Journal of Symbolic Logic* 10.4 (1945). Realizability: a proposition holds iff it has a computable realizer, a program witnessing it., pp. 109–124.

[22] Imre Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery*. The method of proofs and refutations: conjecture, counterexample, lemma-incorporation (restrict the domain by the guilty lemma), retry. Cambridge University Press, 1976.

[23] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Version 8.0. LogiCal Project. 2004. URL: https://coq.inria.fr.

[24] Leonardo de Moura et al. “The Lean Theorem Prover (System Description)”. In: *Automated Deduction – CADE-25*. Springer, 2015, pp. 378–388. DOI: 10.1007/978-3-319-21401-6_26.

[25] Nicolas Bourbaki. *Theory of Sets*. English. Elements of Mathematics. The function-as-graph convention — a function $f : A \to B$ IS the set of pairs $\{(a, f(a)) \mid a \in A\} \subseteq A \times B$ — is the foundational identification the §3 page-2 hinge rests on. Bourbaki §II.3.4 (the definition of a functional graph). Springer, 1968.

[26] Jean-Pierre Marquis. "The Structuralist Mathematical Style: Bourbaki as a Case Study". In: *The Architecture of Modern Mathematics*. Oxford University Press, 2022.

[27] Stephen Cole Kleene. *Introduction to Metamathematics*. Amsterdam: North-Holland, 1952.

[28] Emil L. Post. "Introduction to a General Theory of Elementary Propositions". In: *American Journal of Mathematics* 43.3 (1921), pp. 163–185.

[29] Giuseppe Rosolini. "Continuity and Effectiveness in Topoi". PhD thesis. University of Oxford, 1986.

[30] nLab authors. *nLab — collaborative wiki on category theory, n-categories, and their applications*. ncatlab.org/nlab/show/nLab, accessed 2026-05-02. 2026. URL: https://ncatlab.org/nlab/show/nLab.

[31] Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics 5. New York: Springer-Verlag, 1971.

[32] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991. ISBN: 9780262660716.

[33] Noam Chomsky. "Three Models for the Description of Language". In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 113–124.

[34] Anil Nerode. "Linear Automaton Transformations". In: *Proceedings of the American Mathematical Society* 9.4 (1958). The Myhill–Nerode theorem: the minimal automaton is the canonical normal form of a regular set (a finite-index congruence), so equality and subset are decidable., pp. 541–544.

[35] Eugenio Moggi. "Notions of computation and monads". In: *Information and computation* 93.1 (1991), pp. 55–92.

[36] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Reading, MA: Addison-Wesley, 1995.

[37] Alfred Tarski. "On the Calculus of Relations". In: *The Journal of Symbolic Logic* 6.3 (1941), pp. 73–89. DOI: 10.2307/2268577.

[38] Chris Heunen and Jamie Vicary. *Categories for Quantum Theory: An Introduction*. Vol. 28. Oxford Graduate Texts in Mathematics. Isometry ($f^\dagger f = \mathrm{id}$), unitary ($f^\dagger f = \mathrm{id} \wedge f f^\dagger = \mathrm{id}$), and the dagger-mono reading. Oxford University Press, 2019.

[39] Peter Selinger. "Dagger Compact Closed Categories and Completely Positive Maps". In: *Electronic Notes in Theoretical Computer Science* 170 (2007). Dagger compact closed categories and CPM; the isometry / unitary vocabulary., pp. 139–163. DOI: 10.1016/j.entcs.2006.12.018.

[40] Chris Heunen and Martti Karvonen. "Limits in Dagger Categories". In: *Theory and Applications of Categories* 34.18 (2019), pp. 468–513. URL: http://www.tac.mta.ca/tac/volumes/34/18/34-18.abs.html.

[41] nLab authors. *Dagger Category*. https://ncatlab.org/nlab/show/dagger+category. The correspondence isometry ↔ dagger-mono, unitary ↔ dagger-iso. 2024.

[42] Stanley Burris and H. P. Sankappanavar. *A Course in Universal Algebra*. Vol. 78. Graduate Texts in Mathematics. The Millennium Edition is freely available at https://www.math.uwaterloo.ca/~snburris/htdocs/ualg.html. Springer, 1981. ISBN: 9780387905785.

[43] Martin Hyland and John Power. "The Category Theoretic Understanding of Universal Algebra: Lawvere Theories and Monads". In: *Electronic Notes in Theoretical Computer Science* 172 (2007), pp. 437–458. DOI: 10.1016/j.entcs.2007.02.019.

[44] The mathlib Community. "The Lean Mathematical Library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. 2020, pp. 367–381. DOI: 10.1145/3372885.3373824.

[45] M. F. Atiyah and I. G. Macdonald. *Introduction to Commutative Algebra*. Quotient rings and ideals (Ch. 1); localization / rings of fractions, with the field of fractions as the case $S = A \setminus \{0\}$ (Ch. 3). Addison-Wesley, 1969.

[46] Walter Rudin. *Principles of Mathematical Analysis*. 3rd ed. Construction of $\mathbb{R}$ from $\mathbb{Q}$ by Dedekind cuts (Ch. 1 and Appendix). McGraw-Hill, 1976.

[47] Serge Lang. *Algebra*. 3rd. Vol. 211. Graduate Texts in Mathematics. Revised edition. Springer, 2002. ISBN: 9780387953854. DOI: 10.1007/978-1-4613-0041-0.

[48] Walid Taha and Tim Sheard. "MetaML and Multi-Stage Programming with Explicit Annotations". In: *Theoretical Computer Science* 248.1–2 (2000), pp. 211–242. DOI: 10.1016/S0304-3975(00)00053-0.

[49] Joachim Lambek and Philip J Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge University Press, 1988.

[50] Philip Wadler. "Theorems for free!" In: *Conference on Functional Programming Languages and Computer Architecture*. ACM. 1989, pp. 347–359. DOI: 10.1145/99370.99404.

[51] Jeff Bezanson et al. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671.

[52] Wenzel Jakob. *nanobind: tiny and efficient C++/Python bindings*. https://github.com/wjakob/nanobind. 2022.

[53] Louis Dionne. *Boost.Hana: Your standard library for metaprogramming*. Part of Boost since 1.61. 2016. URL: https://github.com/boostorg/hana.

[54] Hana Dusíková. "Compile-Time Regular Expressions". In: *CppCon*. https://github.com/hanickadot/compile-time-regular-expressions. 2018.

[55] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall International Series in Computer Science. Relational calculus of optimisation: max R = membership intersect (R / converse-membership), the power transpose Lambda, and argmax as max R composed with Lambda F (Ch. 10, Optimisation problems). Grounds the point-free min/max/argmax surface. Prentice Hall, 1997.

[56] Claude Berge. *Topological Spaces: Including a Treatment of Multi-Valued Functions, Vector Spaces and Convexity*. The maximum theorem: the argmax is an upper-hemicontinuous, set-valued correspondence – the classical grounding for argmax-as-a-relation. Oliver and Boyd, 1963.

[57] National Society of Professional Engineers. *NSPE Code of Ethics for Engineers*. Latest revision July 2019. NSPE Publication #1102. 2019. URL: https://www.nspe.org/sites/default/files/resources/pdfs/Ethics/CodeofEthics/NSPECodeofEthicsforEngineers.pdf.

[58] Alan M. Turing. "Systems of Logic Based on Ordinals". In: *Proceedings of the London Mathematical Society*. Second Series 45 (1939), pp. 161–228.

[59] Kazimierz Ajdukiewicz. *Język i poznanie: Wybór pism z lat 1920-1939*. Vol. 1. Foundational text for the Polish School of Logic and Semantic Epistemology. Państwowe Wydawnictwo Naukowe, 1985.

[60] Scott Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. Request for Comments RFC 2119. The canonical normative-keyword vocabulary (MUST, SHOULD, MAY, …). §6 (“Guidance for use of these Imperatives”) stresses careful, sparing application; cited here for prompts to agentic systems where the same restraint applies. Internet Engineering Task Force, 1997. DOI: 10.17487/RFC2119. URL: https://www.rfc-editor.org/rfc/rfc2119.

[61] *<Programming> Journal — Policy on the Use of AI Tools in Submitted Manuscripts*. https://programming-journal.org/purpose/ai.html. Disclosure-of-assistance policy: tools used, tasks, oversight; AI is acknowledged, not co-authored.

A System F, Structuralism, and Jlt

§2 reports the System-F reading as a side effect of the Juliet Posture rather than a design goal: the disciplined subset chosen for alignment with **Alg**(Σ) and clean integration with `std::sem`⌋ `iregular` carriers turns out to be a faithful image of System F (Pierce [4] ch. 23, 29) and of the simply-typed λ-calculus under Lambek–Scott [49], and the same fragment lines up with Bourbaki structuralism [26]. The triangulation was not engineered for; it fell out gratis on the disciplined fragment, which is why it is worth recording. This appendix records the artefact: a row-by-row mapping across the three vocabularies (Table 8), and the admissibility rules an engineer must follow for that mapping to hold (Table 9).

Table 8: The structuralist bridge (Pierce [4] ch. 23, 29; Wadler [50]; Marquis [26]; Bourbaki 1948). Each row is a *correspondence*, not a strict equivalence: the C++ discipline (middle column) recovers the *flavour* of the System F property, exactly for some rows and only in part for others.

System F property	**Juliet Posture (C++)**	**Structuralist style (Bourbaki)**
Parametricity (*Theorems for Free!* [50])	**Concept-gating**: `requires`-clauses fix the interface to a *scent*, so a caller interacts only through the axiom-shape concepts. Parametric use *inside* the body stays a convention, not a compiler guarantee.	**Typification**: ignoring internal set-elements in favour of relations among them.
Universal quantification ($\forall\alpha.\ldots$)	**Constrained templates**: `template<Scent T>` reads as universal quantification over the species defined by the concept *Scent*.	**Species of structure**: the genus of all objects sharing a given structure.
Type erasure (computation is type-blind)	**Zero-cost abstraction**: `std::semiregular` objects with transparent value semantics let the compiler optimise on structure while the program's logic remains independent of the carrier name.	**Indistinguishability**: isomorphic structures are functionally identical.
Strong normalisation (termination)	**Structural pruning**: rejecting Turing-complete template recursion in favour of finite algebraic compositions bounds *type-level* expansion to a decidable cone (and collapses logically empty constructions to zero machine instructions). It is weaker than strong normalisation: over-deep instantiation errors out rather than normalising.	**Echelon construction**: finite, well-founded builds from base sets and axioms.
Impredicativity (types quantified over types)	**Meta-concepts**: concepts defined in terms of other concepts (higher-order *constraint* composition) let the library name 2-arrow structures at the type level. This is an analogy, not System F impredicativity.	**Higher-order structure**: structures built upon structures.
Ad-hoc rejection (no lineage-based dispatch: overload resolution, ADL, specialisation)	**Ban on ADL / specialisation**: ADL beyond textbook operators, public-API specialisation, and SFINAE branching are excluded so that lineage cannot hijack scent (Table 9).	**Axiomatic purity**: rejection of "substance" or non-law behaviour at the level of the structure.

Where the correspondence is partial. Three rows recover the *flavour* of a System F property without the full theorem, and it is worth marking where the analogy stops. *Parametricity* holds at the interface, not inside the body: the admissibility rules (Table 9) exclude ad-hoc dispatch (SFINAE, ADL beyond textbook operators, public specialisation) but not body-level type inspection, so a constrained function may still branch on `std::is_integral_v<T>`; it is a discipline on the fragment, not a property the compiler discharges. *Strong normalisation* is only partial: the bounded-depth rule caps type-level expansion, but over-deep instantiation is a compiler error rather than a normal form, and `constexpr` / runtime recursion is out of scope. *Impredicativity* is weaker still: the C++ analogue is concepts composed from concepts (higher-order constraints) ranging over concrete types (`IsFunctor<F>` takes a `typename F`), not System F's quantification over ∀-types. Each is a structural analogy, not an identity.

Admissibility rules. "Carefully selected" is not informal: the library polices a fixed list of admissibility rules on every `export`-qualified declaration. Each rule closes a specific escape hatch by which C++ would otherwise escape its System-F reading (Table 9); together they delineate the disciplined fragment over which the rosetta of Table 8 holds. The list is not novel: working C++ teams police variants of it informally. The contribution is that the rules are what make the rosetta accurate, and that each rule has a mechanical witness in the partition graph (the public-API headers exported by each `export module` declaration are the audit surface).

Table 9: **The admissibility rules**: which C++ idioms are admitted in the library's public surface, and what each rule rules out. The discipline is the price of the System-F reading; each rule below is the cost of one row of the structuralist-bridge rosetta (Table 8). Internal partitions may use the listed escape hatches in bounded scopes, but they do not appear in any `export`-qualified signature.

Rule on the public surface	What it rules out, and why
No `reinterpret_cast` in any `export`-qualified API.	Defeats type abstraction at the value level. Bit-level reinterpretation (e.g., IEEE-754 inspection) lives in non-exporting internal partitions.
No SFINAE branches in the public surface.	Substitution-failure-as-overload-selection makes the same call mean different things at different call sites. Concept-constrained templates fail with named-axiom messages instead.
No ADL-dependent dispatch except for textbook operator overloads.	Argument-dependent lookup is admitted only where the textbook reads `a + b` or `a == b` on library carriers; free-function calls in user code dispatch through explicit qualification. A library customization point that a type registers beside its own definition (a node's `legs`, a carrier's `structured_and`) is found the same way, by the engine, never by user code.
No template-template parameters in `export`-qualified signatures.	Higher-kinded structure is named at the level of concepts (`IsFunctor<F>`, etc.), not by passing template-templates through public function signatures.
No public-API specialisation that branches behaviour.	Generic templates are parametric across the public surface; per-type specialisations live in the trait registry, each carrying an algebraic claim that is reviewed and CI-checked against a `static_assert`.
Bounded template instantiation depth.	The Turing-complete instantiation machinery is bounded by the carrier-poset depth (scalar, shape, quotient axes; depth ≤ 3 in the current surface), which keeps the type checker's work within a decidable cone.

Three load-bearing threads. Three threads run through the rosetta and stand on their own as the substantive claims of the System-F recovery:

1. **The postural ban on overloading.** System F does not admit ad-hoc polymorphism: overload resolution cannot make a call mean different things for `int` versus `float`. **Jlt** recovers this by banning ADL beyond textbook operator overloads and public-API specialisation. If a function is defined for `Scent T`, it must work the same for every rose that smells like `T`.

2. **Axiomatic honesty.** System F assumes its types follow their definitions perfectly. **Jlt** recovers this by auditing carriers (e.g. rejecting `int` for ring operations where overflow would defeat associativity) so the carrier-rosetta (§E) matches the textbook expectations rather than papering over them.

3. **Transparency over encapsulation.** System F types are *transparent* to the calculus. **Jlt** recovers this by rejecting OO-style encapsulation and private state, so type-directed collapse can see all the way through an object to perform formal reductions.

Taken together, these features give `clang++` a workflow that carries a striking resemblance to a System F type-checker, with optimised machine code falling out as a side effect rather than as a separate compilation step.[11]

An old capability, newly accessible. None of the calculus underneath is new. Templates have always been a System-F-flavoured substrate *within a carefully selected subset of C++*: type-parameter polymorphism, β-style substitution via `using` aliases, normalisation through `constexpr` evaluation. The Lambek–Scott correspondence [49] and the Curry–Howard reading [7] have applied to that fragment for as long as it has existed. What changed in C++20/23 was not the lambda calculus underneath; it was the addition of `concept`, `requires`-clauses, and NTTPs, which together turned a long-standing latent capability into a surface the type checker can discharge mechanically. Practitioners with deep template-metaprogramming experience have known this terrain for over a decade (Boost.Hana [53] and CTRE [54] are the visible end of a long lineage); the contribution of this paper is to walk it under the named-structure / discharged-by-the-compiler discipline, end-to-end, in standard C++23. The translation table above (Table 8) is the dictionary between what was always there and what the modern surface now lets one say out loud.

[11] The well-known alternative to this *disciplined embedding* is the *two-language problem* [51]: research code is written in an expressive high-level language (Python, R, Julia) while only performance-critical execution is delegated to a separately maintained C or C++ backend. Bridging tools such as `nanobind` [52] close the *binary* gap but not the *semantic* gap: the C++ backend cannot reason about high-level mathematical invariants defined in the Python layer. The Juliet Posture is the alternative: keep the high-level mathematical vocabulary in the same language the optimiser sees.

B Certification and Honest Rejection of Carrier Semantics: The Traits Registry

Where Appendix A grounds the structural surface, this appendix expands on how the type system can be used to certify the semantics layer of a carrier, with a special focus on the built-in primitive types and the laws of basic arithmetic. Specifically, it explores the tension between the definitions of the IEEE standard, largely informed by the bounded physical substrate of computation on the one hand, and the abstract definitions of algebra that we aim to model on the other. A fully general, automatic reconciliation is out of reach: deciding whether an arbitrary carrier obeys a given algebraic law is a non-trivial semantic property of its operations, hence undecidable by Rice's theorem. Certification is therefore supplied on a case-by-case basis.

B.1 Three Tiers of Certification

Every law in the tables that follow is graded by *how* the type system knows it holds, on three tiers of increasing strength. *Literal shape* is the weakest: the C++ operators `+`, `-`, `*` close on the carrier directly, so `a + b` compiles and returns the carrier type. *Functor-parametric shape* lifts this to the chosen function objects: `std::plus<T>` and `std::multiplies<T>` return `T` by signature, so the operators close under the functors a generic algorithm actually names. *Strict* is the categorical proof: the carrier satisfies an axiom bundle itself (`concept IsRing`, `concept IsField`), not merely its operator shape. When all three agree under `(std::plus, std::multiplies)`, the bundled `concept IsArithmeticRing` fires and generic code may be written in plain syntax over the carrier. This distinction is necessary because the tiers can disagree: a carrier can

- fail the literal tier yet pass the functor shape. For example: some operators promote `bool` to `int` while the functors indicate closure.
- pass the literal tier (the operators compile) yet fail the strict tier (the axioms do not hold). Example: the IEEE and signed-overflow situation this appendix catalogues.

B.2 Certification and Rejection of Basic Arithmetic

```cpp
// is_associative_v<T, Op> is the structural trait.  Its primary template
// defaults to false: a carrier/op pair is non-associative unless an
// explicit specialisation certifies otherwise.

// Unsigned integrals are certified associative under +  (modular arithmetic).
template <std::unsigned_integral T>
inline constexpr bool is_associative_v<T, std::plus<T>> = true;

// Signed integrals are explicitly REJECTED: signed overflow is undefined
// behaviour, so + is not a law-abiding monoid on signed carriers.
template <std::signed_integral T>
inline constexpr bool is_associative_v<T, std::plus<T>> = false;

// IEEE double needs no specialisation of its own: with no certificate it
// falls to the false-by-default primary, so rounding non-associativity is
// rejected without a per-carrier line.

static_assert( is_associative_v<unsigned int, std::plus<unsigned int>>);  //
↪  certified
static_assert(!is_associative_v<int,          std::plus<int>>);           //
↪  explicit reject
static_assert(!is_associative_v<double,       std::plus<double>>);        //
↪  reject by default
```

Listing 21: The species registry records default algebraic properties (or their absence). `std::unsigned_integral` types such as `std::size_t` are certified `is_associative_v`. `std::signed_integral` types such as `int` are explicitly black-listed due to overflow. `double` is rejected by the false-by-default policy, with no certificate of its own.

Many algebraic laws are certified by the trait registry in `dedekind.category:species` via C++23 template traits. Listing 21 exhibits the certification of associativity of addition, which is a given for additive monoids, specifically $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$:

$$(a + b) + c = a + (b + c) \tag{18}$$

By default, a generic algorithm requiring associativity therefore fails to compile on `int` or `double` without an explicit numeric policy: modular semantics, compensated summation, an interval wrapper, or a lift to an exact field such as $\mathbb{Q}$ or $\mathbb{C}$.

B.3 Algebras of the Primitive Types

Table 1 gives the headline; the four tables below break the principal primitive types down by operator surface, one table per standard-concept family (`bool` on its own), since the built-in primitives share no single operator surface. The same carriers, organised by algebraic structure, are catalogued in the master table of the carrier-poset appendix (Table 14). A ✓marks a law certified in the trait registry, × one that is absent (hence rejected at the gate), and ∘ one that holds only in part.

Table 10: `bool`. The two-element type carries a different algebra under each operator surface.

operator surface	structure
logical	Boolean algebra ✓
bitwise	Galois field $\mathbb{F}_2$ ✓
comparison	total order (false $<$ true) ✓

Table 11: `std::unsigned_integral`. Modular arithmetic gives a genuine ring; the narrow types leak through integer promotion.

type(s)	arithmetic	bitwise	comparison
`unsigned int`, `unsigned long`, `unsigned long long`, `size_t`	ring $\mathbb{Z}/2^n$ ✓	Boolean algebra ✓	total order ✓
`unsigned short`, `unsigned char`	ring $\mathbb{Z}/2^n$ ∘ (promotes)	Boolean algebra ✓	total order ✓

Table 12: `std::signed_integral`. Signed-overflow UB denies the ring; the two's-complement bits (guaranteed since C++20) and the order survive.

type(s)	arithmetic	bitwise	comparison
`int`, `long`, `long long`	partial × (overflow UB)	Boolean algebra ✓	total order ✓
`signed char`, `short`	partial × (UB, promotes)	Boolean algebra ✓	total order ✓

Table 13: **std::floating_point.** On IEC 60559 (IEEE-754) targets, rounding defeats associativity and NaN defeats even the order; almost nothing survives.

type(s)	arithmetic	comparison
float, **double**, **long double**	closure only × (rounding)	not a total order × (NaN incomparable)

C An Elementary Theory of the Category of Sets

This appendix details the step-by-step ETCS embedding that Section 3 summarises: Lawvere's ten axioms reified as the composed type-predicate `concept IsSet`.

A common implementation approach follows material Zermelo-Fraenkel (ZF) set theory, where the discussion starts from an element membership relation $\in$ acting on a cumulative universe of sets. Implementations such as `std::set` use this as a starting point and model sets as *extensional memory containers*. The identity of a set is determined by explicitly enumerating its members in physical allocations, and membership is evaluated by traversing a data structure (e.g., a balanced binary search tree). This approach exhibits prohibitive shortcomings when applied to algebraic and engineering problems: for example, it cannot natively model infinite sets like the natural numbers $\mathbb{N}$, or non-enumerable continuous intervals over the real numbers $\mathbb{R}$. As a result, both the structural guarantees a compiler can infer and the optimizations it can perform on such problems are severely restricted.

By contrast, the *structural categorical comprehension* view pioneered by William Lawvere [5] abandons element-centric containment. Instead, it defines a set entirely through its relational behavior using objects and functions (*morphisms*). In this framework, the question is not what an element *is* internally, but how it behaves under functional composition. Lawvere's Elementary Theory of the Category of Sets (**ETCS**) posits that the category of sets (**Set**) is a well-pointed, Cartesian Closed Category with a subobject classifier, a natural numbers object, and the axiom of choice.

To bridge this abstract mathematical formalism with systems-programming primitives, the foundational axioms of ETCS are translated into explicit C++23 type-predicates (`concept IsSet`) distributed across six localized compile-time partitions:

C.1 Small Categories (`:small` partition)

- **Axioms of Categories & Smallness:**
  1. *Category:* a collection of objects and arrows (morphisms) closed under an associative composition with identities.
  2. *Smallness:* both collections are *sets*, not proper classes. This holds *by construction*, since the objects and arrows are the value-sets of single C++ types (`Cat::Species` and `Cat::Arrow`), and a type's value-set is a set in the metatheory. (Requiring only the hom-collection to be a set gives the weaker notion of a *locally small* category.)

- **C++ Reification:** The `concept IsSmallCategory` gates the structure. Per **Jlt** 2, objects are `std::regular` carriers. Morphisms are values of a single arrow type satisfying `concept IsArrow`, with `::Domain` and `::Codomain` typedefs and a callable `f(x)` from the domain into the codomain. Composition is the overloaded `operator>>`, with the identity laws `id >> g == g` and `f >> id == f` discharged as overloads; identity arrows come from an `id_c(x)` factory.

- **Explanatory Note:** Instead of data-hiding containers, a "set" at this entry rung is simply a well-behaved C++ type acting as a domain, and a "function" is a side-effect-free arrow mapping inputs to outputs. A carrier taken with its endomorphisms is a one-object category, a **monoid**; this is the ambient **Set** witnessed at that carrier, not a claim that an individual set is itself a category. An individual set is an *object* of **Set** [5]; it presents *as* a category only under the discrete embedding $\mathrm{Disc}\colon \mathbf{Set} \hookrightarrow \mathbf{Cat}$, which sends it to the discrete category on its elements. Accordingly `IsSet` requires the categorical axioms of `CanonicalSetCCC<A::Domain>`, the **Set**-witness over the carrier, and never asks the set type itself to satisfy `IsSmallCategory`.

C.2 Cartesian Categories (`:limit` partition)

- **Axioms of Terminal Object & Finite Products:**
  1. *Terminal Object:* There is a terminal object 1, characterised purely by its universal property: every set A has exactly one arrow $A \to 1$.
  2. *Binary Products:* For any sets A and B there is a product $A \times B$ with projections $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$, universal in that every pair $f : X \to A$, $g : X \to B$ factors through a unique $\langle f, g \rangle : X \to A \times B$.
- **C++ Reification:** `concept IsTerminalObject` and `concept IsProduct` gate the structure, and the standard library satisfies them out of the box: `std::monostate` *is* the terminal object (exported as `One`), and `std::pair` *is* the binary product, with projections π_1/π_2 reading `.first` / `.second`. The universal factorisation $\langle f, g \rangle$ is the one piece `std` does not provide; the library supplies it as the default `mediate_product` arrow.
- **Explanatory Note:** This layer turns relational coordinate tables into type-safe tuples. The terminal object lets us define "elements" of a set A structurally, as arrows `One` $\to A$.

C.3 Cartesian Closed Categories (`:cartesian` partition)

- **Axiom of Exponentials:** For any sets A and B there is an exponential B^A: the set of functions $A \to B$ with an evaluation map $\text{eval} : B^A \times A \to B$, universal in that every $f : X \times A \to B$ factors through a unique $\text{curry}(f) : X \to B^A$ (the adjunction $(- \times A) \dashv (-)^A$).
- **C++ Reification:** The exponential B^A is the internal hom, whose inhabitants are the functions $A \to B$. It is required by `concept IsCartesianClosed` via `concept IsExponential`. The adjunction arrows `curry` / `uncurry` / `eval` are library-provided.
- **Explanatory Note:** This licenses function *Currying* at compile time via the `constexpr` lambdas. It guarantees that functions are first-class citizens that can be passed, stored, and transformed by other functions without dropping out of the type-checking engine.

C.4 Elementary Topoi (`:topoi` partition)

- **Axioms of Subobject Classifier & Equalizers:**
  1. *Subobject Classifier:* a truth-value object Ω with an element $\text{true} : 1 \to \Omega$ such that every mono $S \hookrightarrow A$ arises as the pullback of true along a *unique* characteristic map $\chi : A \to \Omega$.
  2. *Equalizers:* for parallel arrows $f, g : A \to B$, the subobject $\{x \in A \mid f(x) = g(x)\}$.
- **C++ Reification:** The ambient classifier Ω is the codomain of a carrier's (possibly partial) characteristic map, fixed by a *logic species* `L` (the code names it `L::Ω`). Its Boolean fragment is the *dominance* $\Sigma \hookrightarrow \Omega$, a Rosolini dominance [29] (Equation 3); strict ETCS is the Boolean collapse $\Sigma = \Omega$. `Boole` gives the Boolean $\mathbb{B} = \Sigma$ (`bool`), a total map; `Kleene` gives the lifted Boolean (`Ternary`, $\mathbb{B} + \mathrm{U}$, under strong-Kleene connectives [27]), the codomain of a partial χ. `concept IsOckhamAlgebra` gates such a species; the predicate of a `Comprehension<𝔸<T, L>, P>` is χ, and `concept IsSubobject` recognises it. Which codomain a carrier receives is the decidability split of Section 3: $\mathbb{B}$ where membership is decidable, the lifted Boolean where it is not. Equalizers are named by `concept IsEqualizer`, the subobject solving $f = g$.
- **Explanatory Note:** An *intensional* set is a rule, not an array. Testing $x \in S$ evaluates $\chi(x)$ instead of searching memory.

C.5 Concrete Topoi (`:concrete` partition)

- **Axioms of the Powerset Monad & Elementality:** Every set A maps functorially to its powerset $\mathcal{P}(A)$ via a unit morphism $\eta_A : A \to \mathcal{P}(A)$.
- **C++ Reification:** Expressed via the `SetAsProduct` layout template, which combines a raw underlying C++ data type primitive with its corresponding logical evaluator lambda.
- **Explanatory Note:** This layer implements a faithful *forgetful functor* U, which strips away structural categorical formatting to expose underlying raw C++ hardware types when evaluation is explicitly required. Constructing lazy evaluation pipelines (e.g., using `std::views::transform`) functions as monadic composition, chaining transformations without intermediate memory allocations.

Axiom 9: Power Set / Exponential Restriction ($\beth_1$). For any set X, there exists an object $\mathcal{P}(X)$ classifying its subobjects. By Cantor's theorem (Lawvere's Fixed-Point Theorem), this explicitly bounds the cardinality of classical constructible sets in the base theory below the second transfinite cardinal. Russell's paradox is ruled out by construction. Library Reification: Restricts carrier predicates from modeling uncountably infinite domains beyond the continuum $\beth_1$, avoiding type-system divergence and ensuring all logical intersections resolve during template translation.

C.6 Set-Modeling Topoi (`:etcs` partition)

- **Axioms of Well-Pointedness, Infinity, and Choice:**
  1. *Well-Pointedness:* If two functions $f, g : A \to B$ are distinct, there must exist a single element $x : 1 \to A$ such that $f(x) \neq g(x)$.
  2. *Infinity:* There exists a Natural Numbers Object (NNO) $\mathbb{N}$ that enables primitive recursion without structural loops.
  3. *Choice:* Every surjective function has a right inverse section.
- **C++ Reification:** Enforced via the `IsSet` concept constraint. Well-pointedness enforces referential transparency and forbids hidden mutable states. Infinity is preserved by generating strict element-level partial orderings via `std::three_way_comparable` rather than managing physical loop limits.
- **Explanatory Note:** The NNO lets the library model *transfinite spaces* (e.g., continuous mathematical boundaries or infinite sequence generators) lazily. The Axiom of Choice (Axiom 10) is only *approximated* here, through the power-object lattice, and remains aspirational: `IsSet` ends in `HasAxiom10PowerObjectLattice`, not a choice/section witness. This matches the main text's undecidable-by-default reading (§3): `IsSet` does not assume Choice, so membership is not assumed decidable and the compiler does not constructively discharge arbitrary witness selections.

D Point-Free Morphisms: Relations and Factorization Gates

Dual to §4, this appendix collects the supporting detail for the point-free morphism layer: relations presented as Kleisli arrows over the powerset monad, and the epic–monic factorization gates ($A \twoheadrightarrow I \hookrightarrow B$) that make a general function's codomain queries decidable. It is another instance of the paper's recurring seam. General first-order quantification over open intensional sets is undecidable (Rice's theorem), so rather than attempt the aspirational universal search the library supplies mechanical support for the *existential* case: lazy Kleisli predicate transformation for composition, and a developer-supplied factorization for image membership. The worked internals are deferred to a follow-up revision.

The optimisation operators of §4 (min, max, arg max) are the relational calculus of Bird and de Moor [55]. On a codomain preorder R, the maximum is the point-free $\max R = (\in) \cap (R/\ni)$: membership $\in$ intersected with the relational *division* (residual) $R/\ni$, which is the point-free spelling of the universal quantifier, and $\arg\max = \max R \cdot \Lambda F$ selects the R-optimal element of the fibre generated by the power transpose $\Lambda F : A \to \mathcal{P}(B)$ (the `apply` of §4). Because R is only a *preorder* (reflexive and transitive, not antisymmetric), distinct elements that are R-equivalent tie for the top and are held by the relation rather than collapsed to one, so $\max R$ is genuinely set-valued: argmax as the set-valued correspondence of Berge's maximum theorem [56]. (The Pareto front, by contrast, is the *maximal* elements, a distinct operator: $\max R$ selects the *greatest*, so it is empty where the front has several incomparable points.) Decidability is the same seam once more: the division $R/\ni$ is a bounded universal, decided by structure (a finite carrier ($\mathbb{B}$), a halfspace pivot on $\mathbb{N}$ or $\mathbb{Z}$, or a finite quotient) rather than by search, so no finiteness is required (the least element of a halfspace is read from its pivot, not enumerated).

E The Carrier Poset and its Realisations

This reference is indexed by algebraic structure (the poset of §5), not by carrier. To answer "I need structure X", read its row in Table 14; to place a type you already hold, scan the realisation columns, or the by-operator-surface tables of §B.3.

Table 14: **The carrier poset and its realisations.** Each structure of the poset (§5), the concept that certifies it, and the carriers that realise it on the built-in (`std` / IEEE) side and the exact-library side. Each row names *representative* carriers that strictly realise the structure; a carrier recurs across the rows it satisfies (`unsigned` is a magma, monoid, group, and ring; `bool` is both a field and a Galois field). † marks operator-surface-only certification, the strict axiomatic witness being queued. `I` abbreviates `default_integer`, the saturating $\mathbb{Z}$ (`SignedCardinality`); the wrapping word `SignedExtensionalCardinal<>` is $\mathbb{Z}/2^w\mathbb{Z}$, certified as such and not as a backing for $\mathbb{Q}$.

Structure	Concept	Built-in realisation	Exact-library carrier
Magma	`IsMagma`	`unsigned` under $+$ / $*$ (modular)	any closed carrier
Comm. monoid	`IsCommutativeMonoid`	`unsigned` under $+$ ($\mathbb{Z}/2^n$)	$\mathbb{N}$: `Cardinality`
Abelian group	`IsAbelianGroup`	`unsigned` under $+$ (cyclic)	$\mathbb{Z}$: `SignedExtensionalCardinal<>`
Ring	`IsRing`	`unsigned` under $(+,*)$; signed `int` *refused* (overflow UB)	$\mathbb{Z}$: `SignedExtensionalCardinal<>`; `Polynomial<R, M>`
Field	`IsField`	`bool` under $(\oplus,\wedge)$ ($\mathbb{F}_2$)	$\mathbb{Q}$: `Rational<I>`; $\mathbb{C}$: `Complex<Rational<I>>`†; $\mathbb{R}$: `QuadraticReal<2>`
Galois field	`IsGaloisField`	`bool` ($\mathbb{F}_2$, order 2)	`F64` (order 64); `Modular<p>` for prime p, e.g. $\mathbb{F}_{\mathtt{p}}$ ($p = 2^{64} - 59$)
Module / v. space	`IsModule` / `IsVectorSpace`	`uint64_t` under $\oplus$, `bool` scalars ($\mathbb{F}_2^{64}$)	`Vec2V<Rational<I>>`; $\mathbb{D}$: `Dual<Rational<I>>`

† Operator-surface certification only (`HasRingOperators` / `HasFieldOperators`); the vector-space lift of §5.2 is clean for $\mathbb{Q}$ and withheld for $\mathbb{C}$ (the field certificate is false over $\mathbb{F}_2$, where $x^2 + 1 = (x+1)^2$). `Dual` carries ring operators, but $\varepsilon^2 = 0$ leaves it without a strict `IsRing` witness, so it sits with the modules.

E.1 How the certificates lift

This appendix is the deep dive behind §5.2: the HSP combinators of Listing 19 put to work across all three families and both lift rules. Registering `Vec2V` is the single line `template <typename T> struct product_algebra_base<Vec2V<T>> { using type = T; };`, which makes `T` the base: `T`'s species traits carry forward to `Vec2V<T>`. So whenever `T` is a ring, `IsModule<Vec2V<T>, T>` follows once the ring, identity, inverse, and action witnesses hold, and Rule 1 upgrades it to `IsVectorSpace` over any field. A synthetic base makes the closure visible end to end (Listing 22): three markers stand for the H, S, and P images of $\mathbb{Z}/6\mathbb{Z}$, and the variety "associative and commutative $+$" propagates to all three, witnessed by eight `static_assert`s.

The quotient combinator carries the dual numbers all the way to a faithful matrix embedding (Listing 23): $\mathbb{D}$ sits inside $M_2(\mathbb{Q})$ as 2×2 matrices, with the infinitesimal ε squaring to zero.

Re-enabling refused carriers. The Honest Rejection policy refuses carriers like `int` as a witness for *axiomatic* ring laws (signed-overflow UB defeats closure as a stable axiom, per Listing 21). An engineer with a local proof that overflow cannot occur on their inputs has two sanctioned re-entry paths:

```cpp
// --- Synthetic HSP markers wired to Modular<6> ----------------------
struct H_image_of_Z6 {};   // type-level placeholder for any H-image of Z/6Z
struct S_subalg_of_Z6 {};  // ... S-subalgebra ...
struct P_product_of_Z6 {}; // ... P-direct-product ...

template <> struct quotient_algebra_base<H_image_of_Z6> { using type =
↪  Modular<6>; };
template <> struct subalgebra_base      <S_subalg_of_Z6> { using type =
↪  Modular<6>; };
template <> struct product_algebra_base <P_product_of_Z6>{ using type =
↪  Modular<6>; };

// --- THE Birkhoff HSP-closed crown ----------------------------------
// Variety V: carriers with associative + AND commutative +.
// V is HSP-closed by the propagation specs.

using Z6 = Modular<6>;  using H = H_image_of_Z6;
using S  = S_subalg_of_Z6;  using P = P_product_of_Z6;

static_assert(is_associative_v<Z6, std::plus<Z6>>);  // base ∈ V
static_assert(is_associative_v<H,  std::plus<H>>);   // H closure (quotient)
static_assert(is_associative_v<S,  std::plus<S>>);   // S closure (subalgebra)
static_assert(is_associative_v<P,  std::plus<P>>);   // P closure (product)
static_assert(is_commutative_v<Z6, std::plus<Z6>>);  // (same four, both axioms)
static_assert(is_commutative_v<H,  std::plus<H>>);
static_assert(is_commutative_v<S,  std::plus<S>>);
static_assert(is_commutative_v<P,  std::plus<P>>);
```

Listing 22: Birkhoff's HSP closure typed. Base $\mathbb{Z}/6\mathbb{Z}$ + three synthetic HSP markers wired to the carrier-side registries; the trait propagation lifts both axioms of the variety uniformly to all three HSP-derived carriers. Any future H/S/P construction over $\mathbb{Z}/6\mathbb{Z}$ (or any other base in the variety) follows the same one-line opt-in pattern.

- *Operator-surface tier.* `concept HasRingOperators<int>` fires unconditionally; downstream combinators that bind to this weaker concept (rather than to the strict `concept IsRing` bundle) admit `int` directly. Most paper showcases bind to the operator-surface tier for exactly this reason.
- *Opt-in trait specialisation.* Where a stronger axiom is needed, the relevant species trait is specialised explicitly, e.g. `template<> inline constexpr bool dedekind::category::is_associative_v<MyType, std::plus<>> = true;`, for a user-defined type whose addition is provably associative. The implementor carries the proof obligation [57]; the library trusts the trait.

The mechanism is intentionally narrow: the engineer can re-enable a specific (carrier, axiom) pair, but cannot re-enable an entire concept bundle without specialising each axiom trait individually. Pedantic by default; opt-in by deliberate engineer action.

Suggested pattern. Constrain generic numeric code on the algebraic concept, not the carrier. One signature bounded by `HasSemiringOperators<T>` admits a machine primitive (`unsigned`) and an exact carrier (`Cardinality`) alike, so the same algorithm compiles for both. The strict `IsSemiring<T>` is where an unsound carrier is turned away: `int` under `+` fails it, signed overflow breaking associativity. The crossings (Table 15) are then the one place a caller deliberately steps off the concept onto a concrete machine type.

```cpp
using Rat = Rational<SignedExtensionalCardinal<>>;
constexpr Dual<Rat> d{Rat{2}, Rat{3}}, e{Rat{-1}, Rat{5}};
static_assert(as_matrix2x2(d + e) == as_matrix2x2(d) + as_matrix2x2(e));
static_assert(as_matrix2x2(d * e) == as_matrix2x2(d) * as_matrix2x2(e));
constexpr Dual<Rat> eps{Rat{0}, Rat{1}};
static_assert(as_matrix2x2(eps) * as_matrix2x2(eps) == zero_matrix2x2_v<Rat>);
```

Listing 23: Dual numbers embed faithfully as 2×2 matrices over $\mathbb{Q}$, with the infinitesimal squaring to zero.

Table 15: **Crossings state-machine.** The named morphisms that move between exact and primitive sides, per role. Each row shows *realise* (exact → primitive: drop exactness for speed) and *embed* (primitive → exact: lift a machine value into the certified carrier). A dash marks a direction that requires going through an intermediate role.

Role	Realise: exact → primitive	Embed: primitive → exact
$\mathbb{B}$	(identity)	(identity)
$\mathbb{N}$	`.realize_to_size_t(sentinel)`	`embed_unsigned_integral<N>(v)`
$\mathbb{Z}$	`static_cast<int64_t>(x)`	`embed_signed_integral<Z>(v)`
$\mathbb{Q}$	(none; no direct $\mathbb{Q}$ → primitive arrow)	`embed_ℒ_ℚ<I>(v)`
$\mathbb{R}$ symbolic	evaluate cut at a rational bound	(intensional; no simple lift)
$\mathbb{R}$ policy	`assume_ieee(ℒ<double>)`	`discharge_ieee(IEEE<double>)`, partial: `std::optional`
$\mathbb{C}$	(none; per-component, no direct arrow)	`Complex<R>(re, im)`
$\mathbb{D}$ (dual)	`.val` (primal only; tangent discarded)	`Dual<F>(v, d)`

F From Cardinality Pruning to Type-Directed Collapse

Step back from the showcases for a moment. What is the pattern? In every one of them we started with an intensional set (a predicate) and we ended with a named extensional object whose *type* carries the answer: $\varnothing$, a singleton, an order interval of known cardinality, an LP vertex. The collapse is not incidental; it is a rewrite rule, running inside the type system, at translation time. To make the claim precise, we first formalise "how much the compiler knows about a set" as a lattice, then state the theorem as monotone motion up that lattice.

Definition 6 (Computability tier)**.** *For a compile-time set S over an ambient carrier T, the* computability tier *is the triple*

$$\mathcal{T}(S) \;=\; \big(\mathsf{R}(S),\ \mathsf{L}(S),\ \mathsf{K}(S)\big) \;\in\; \{\mathrm{Int},\mathrm{Ext}\}\times\{\mathrm{K3},\mathrm{Cl}\}\times\{\mathrm{Opq},\mathrm{Elt}\}$$

whose components are the representation *tier (R: intensional vs. extensional, i.e. predicate vs. elements-in-type), the* logic *tier (L: Kleene three-valued vs. classical two-valued membership), and the* knowledge *tier (K: elements opaque vs. carried in the type). Each axis carries the order* $\mathrm{Int}\preceq\mathrm{Ext}$, $\mathrm{K3}\preceq\mathrm{Cl}$, $\mathrm{Opq}\preceq\mathrm{Elt}$*; the product order is componentwise. The resulting Boolean lattice $(\mathcal{T},\preceq)$ has bottom $\bot=(\mathrm{Int},\mathrm{K3},\mathrm{Opq})$ ("a raw predicate over a floating-point carrier with no inhabitants known") and top $\top=(\mathrm{Ext},\mathrm{Cl},\mathrm{Elt})$ ("a named finite set whose members are template parameters and whose membership is decidable").*

Every showcase of Section 3.3 is a motion inside this lattice. Showcases 3 and 4 both act over $\mathbb{N}$, whose membership is already classical, so they start at $(\mathrm{Int},\mathrm{Cl},\mathrm{Opq})$, not at $\bot$. Their reductions move up the knowledge axis in one step: the empty-halfspace meet of Showcase 3 lands on $\varnothing$, and the cardinality-1 meet of Showcase 4 lands on the point $\{4\}$, an element known as a value. The representation axis does not move: the element rides in the value, not in a template parameter, by design, so that runtime pivots reduce through the same law. The logic axis is already at Cl and does not move; a genuine $\mathrm{K3}\mapsto\mathrm{Cl}$ (restoring decidability) needs a carrier whose membership is three-valued, not a halfspace on $\mathbb{N}$. The reductions never slide downward. With that in hand:

Theorem 1 (Collapse of a halfspace meet, totally-ordered 1D case)**.** *Let T be a totally ordered carrier, and let*

$$P_i(x) \;=\; (x \ge a_i),\quad i=1,\dots,m, \qquad Q_j(x) \;=\; (x \le b_j),\quad j=1,\dots,n,$$

be lower-bound and upper-bound halfspace predicates over T whose pivots $a_i, b_j \in T$ are constants. Define the effective bounds $a^\star := \max_i a_i$ if $m \ge 1$ and $a^\star := \bot$ if $m = 0$, and $b^\star := \min_j b_j$ if $n \ge 1$ and $b^\star := \top$ if $n = 0$ (where $\bot, \top$ denote the carrier's bottom and top elements, taken as $\pm\infty$-style sentinels when T is unbounded). Then the meet

$$S \;=\; \bigcap_{i=1}^{m}\{x\in T \mid P_i(x)\} \;\cap\; \bigcap_{j=1}^{n}\{x\in T \mid Q_j(x)\}$$

reduces, in constant evaluation, to exactly one value of `Interval<T>`*:*

1. *$\varnothing$, when $a^\star > b^\star$;*
2. *the point $\{v\}$, when T is integral and $a^\star = b^\star = v$;*
3. *the interval $[a^\star, b^\star]$ otherwise.*

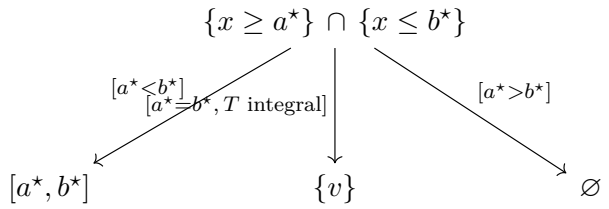


Figure 4: The 1D collapse as a diagram chase (Theorem 1). The meet of the effective lower- and upper-bound halfspaces reduces to exactly one value of `Interval<T>`, selected by the side condition on the pivots: the interval $[a^\star, b^\star]$ for the generic bounded case ($a^\star < b^\star$); the point $\{v\}$ for the degenerate equality case ($a^\star = b^\star = v$) *when T is integral* (on a non-integral T the equality case stays the degenerate interval $[a^\star, a^\star]$); and $\varnothing$ for the contradiction case ($a^\star > b^\star$). The diagram commutes on elements: x inhabits the source meet iff x inhabits the value the side condition selects. Implementation hook: the meet is `reduce_meet` in `dedekind.order:halfspace`, the tighter bound on each side followed by one collapse law; it is `constexpr`, so constant pivots fold at compile time and runtime pivots reduce at runtime. Mechanically witnessed by Showcase 3 (the empty branch, `showcase_03_halfspace_contradiction.cpp`) and Showcase 4 (the point branch: on the integral carrier $\mathbb{N}$ the strict halfspaces $x > 3$, $x < 5$ normalise to $x \geq 4$, $x \leq 4$, so this is the $a^\star = b^\star = 4$ case of Theorem 1); both fold to constant IR returns under `-O2`, as the bottom panel of Listing 3 demonstrates for the empty-meet branch.

Proof. By induction on $m + n$.

Base case ($m + n = 0$): no constraints; $S = T$, which is the order interval $[\bot, \top]$ (case 3) by the boundary convention.

Inductive step. Assume the claim holds for k total constraints. Add one more; without loss of generality (by the symmetry between lower and upper bounds) take it to be a fresh lower-bound predicate $P_*(x) = (x \geq a_*)$. The inductive hypothesis gives a reduced form S_k of one of the three shapes. Form $S = S_k \cap \{x \mid x \geq a_*\}$. Two compile-time comparisons of a_* against the bounds of S_k (where present) suffice:

- **Redundant.** If S_k has lower bound a and $a_* \leq a$, then $\{x \mid x \geq a_*\} \supseteq S_k$ and $S = S_k$; the form is preserved.
- **Tightening.** If S_k is the interval $[a, b]$ (or unbounded above) and $a < a_* \leq b$ (or no upper bound), then $S = [a_*, b]$ in case 3; if T is integral and $a_* = b$, then $S = \{a_*\}$ (case 2).
- **Contradicting.** If S_k has upper bound b and $a_* > b$, or if S_k is already $\varnothing$, then $S = \varnothing$ (case 1).

The same case-split, with rôles of a_i and b_j exchanged, discharges the symmetric upper-bound addition. Each case lands in one of the three forms, completing the induction. □

Theorems for free. The structural shape of Theorem 1 is Wadler's *Theorems for Free!* [50] move at the algebraic register. Wadler showed that polymorphic types in the typed lambda calculus discharge a class of theorems by parametricity alone, with no further proof obligation on the caller. Here, the analogous discharge runs from *carrier-axiom declaration* (`HasTotalO` ⌟ `rderOperators<T>` + `IsTotallyOrdered<T>`, plus the integral-carrier witness when case 2 is to fire) to a carrier-level reduction of any finite intersection of half-line constraints. The engineer's honesty obligation is the axiom set; the collapse is type-level free. The proof above is one demonstration that the discharge is mechanical: the inductive argument does not depend on which carrier or which pivot type, only on the axiom slots being filled.

Existential anchor. The theorem is realised in the library by witnesses that materialise at the static-assertion layer; the canonical exhibit on $\mathbb{N}$ is the collapse

$$\{x \in \mathbb{N} \mid x \geq 2\} \cap \{x \in \mathbb{N} \mid x \leq 5\} \longrightarrow [2, 5],$$

checked at compile time. Section 3.3 catalogues these witnesses; the LLVM IR fixtures pin the resulting machine-code collapse downstream of the type-level reduction.

Conjecture 1 (Type-Directed Collapse, n-dimensional generalisation)**.** *Let $T_1, \ldots, T_d$ be totally ordered carriers and let $\{P_k\}_{k=1}^{N}$ be halfspace predicates over the product $T_1 \times \cdots \times T_d$ whose pivots are carried at the type level. The meet*

$$S = \bigcap_{k=1}^{N} \{(x_1, \ldots, x_d) \mid P_k(x_1, \ldots, x_d)\}$$

admits a compile-time reduction to a typed normal form generalising the three cases of Theorem 1 (empty, vertex-singleton, or a polytope record carrying its active constraint set), under a suitable axiom set on the product carrier (total order per axis, closure of the meet under the product, and a vertex-enumeration discipline for the polytope case). Writing S_k for the single-constraint set, the reduction is monotone up the computability-tier lattice:

$$\mathcal{T}(S) \succeq \bigvee_{k} \mathcal{T}(S_k),$$

and strictly so on at least one axis in every case the library exhibits.

Status of claim. *This conjecture is* not *proved in the paper. It is supported by the same exhibition pattern as the 1D theorem: each case is checked into the library as a* `static_assert/STATIC_CHECK` *witness and, where relevant, as an LLVM IR fixture that fails CI on optimiser drift (see Section 3.3). A general inductive proof over dimensions d is left open.*

Corollary 1 (Iterated collapse)**.** *Any finite composition of type-directed-collapse reductions is itself monotone up $\mathcal{T}$. In particular, no sequence of library operations can ever return a set that is* less *decidable than its inputs.*

So: not "a faster way to do set algebra", but a rewrite rule that happens to live in the C++ type system, and whose action is monotone-ascending in a lattice of decidability. The halfspace case is the smallest non-trivial instance we know how to prove mechanically. Three further axes appear to obey the same pattern; the project tracker carries one existential proof per axis (one worked instance, one IR fixture, one compile-time witness) rather than a general framework. We would rather ship one reduction that fully collapses than four sketches that do not.

- **Carrier tightening.** Intersecting a subset of $\mathbb{N}$ with a subset of $\mathbb{R}$ should land in $\mathbb{N}$: the carrier poset contributes its own structural reduction via the existing canonical embeddings (`embed_uint_sint_`, `embed_ℚ_ℝ`, …).
- **Smooth optimization.** The stationary set $\{x \mid \nabla f(x) = 0\}$ for a quadratic f collapses to the point $\{x^\star\}$, the minimizer carried as a value. Same reduction pattern, applied at the extremum operator. *Scope note:* the polytope / linear-programming domain is not incidental. Active-set enumeration over halfspaces is constructively bounded at compile time precisely because halfspaces have finite combinatorial structure; non-polyhedral domains (KKT-stationary-point search over arbitrary smooth f) require symbolic root-finding which does not terminate at template- instantiation time. Quadratic f is the worked-instance boundary; non-linear extensions are post-paper work tracked under #363.

- **Lattice-law rewriting.** Distributivity and absorption applied at the type level rewrite $(A \cup B) \cap \neg A$ to $B \cap \neg A$ *before* local reduction rules fire, exposing collapses that a purely local analysis would miss.

Each axis sits on the foundation shared with Section 3.3: NTTP predicates, structured `operator&` dispatch, and the computability tiers visible in the showcases. Taken together they sketch a program of which the present halfspace reductions are one installment.

The polynomial-calculus exhibit (companion-repository `:polynomial` / `:ftc` partitions) belongs to the same family but at the *function* layer: a polynomial reduced under `derive` (a ring homomorphism) or a numerical integrator yields another intensional object whose identity with a target is discharged by the compiler.

Scaling beyond atomic carriers via structure-preserving combinators. The same partition (atoms below, runtime above) governs the linear-algebra surface the paper exhibits at the 2×2 scale. The library ships `Invertible2x2`, `Matrix2x2V`, `Vec2V` as statically-validated atoms; larger systems are *not* reached by inflating the atom (we do not aim for $n \times n$ inversion in templates), but by composing atoms through *structure-preserving combinators* that propagate the invariants up. `DirectSum<A, B>` carries invertibility from its factors to the block-diagonal whole and is shipped today; 4×4 invertibility is therefore already a one-combinator step from the 2×2 atom, with $(A \oplus B)^{-1} = A^{-1} \oplus B^{-1}$ exact over $\mathbb{Q}$ and verifiable by `static_assert`:

```
TEST_CASE("linear_algebra:matrix --- Tier 1: "
          "DirectSum preserves invertibility") {
  constexpr Invertible2x2<Rat, Rat{1L}, Rat{2L},
                          Rat{3L}, Rat{4L}> block_M{};
  // 90° rotation over ℚ.
  constexpr Invertible2x2<Rat, Rat{0L}, Rat{-1L},
                          Rat{1L}, Rat{0L}> block_rot{};
  constexpr DirectSum<decltype(block_M),
                      decltype(block_rot)> ds{};
  constexpr auto ds_inv = ds.inverse();

  using IdentityDirectSum =
      DirectSum<Identity2x2<Rat>, Identity2x2<Rat>>;
  STATIC_CHECK(ds * ds_inv == IdentityDirectSum{});
  STATIC_CHECK(ds_inv * ds == IdentityDirectSum{});
}
```

`BlockUpperTriangular<A, B, D>` extends the same propagation to triangular blocks under a degenerate-Schur condition (also shipped, also testable as a single `static_assert`); a future Kronecker combinator preserves orthogonality and lattice structure compositionally. Each combinator is itself a small structurally-checked unit; their composition gives $2^k \times 2^k$ matrices whose invariants follow by induction without exponential blowup in the templates. The remaining roadmap items fall out of the same Axiomatic-Systems-Programming discipline rather than requiring a different paradigm, and are tracked in the project backlog rather than discharged here. The $2 \times 2 \to 4 \times 4$ step is a witness, not a promise; everything beyond is the natural recursion of the same combinator pattern.

Carrier choices, error messages, ergonomics. The carrier-vs-overflow trade-offs are catalogued in Section E's Rosetta tables: showcases default to `Rational<SignedExtensionalCardinal<>>` (a structural carrier, usable as a template argument; it wraps at 2^{63}, so it is certified on its operator surface only, the strict field being `Rational<default_integer>`); programmers who trade exactness for native performance pick a primitive row and accept the named failure

mode (signed-overflow UB on `int`; non-associative rounding on `float`/`double`). The combinators of Section 3.3 bind to `HasRingOperators` rather than any particular carrier, so the swap is a one-line change at the `using` declaration. Error messages, by contrast, are a more universal C++ pain: a failed `static_assert` in a deep reduction can produce a multi-page expansion trail that no library can fully repair. The realistic ambition is to *fail fast at the named axiom*: the first message the reader sees should be the predicate that genuinely broke, not the cascade of downstream failures it triggered. The library structures concept ladders so that each rung adds a single named requirement, and uses targeted `static_assert` messages inside internal reductions, so a violation surfaces at the smallest witness rather than deep inside overload resolution.[12]

In short: the compiler wall has a predictable shape (depth, steps, combinatorics, numeric range, diagnostics), and each is either pushable by a flag, avoidable by staying within the instance regime for which this technique is intended, or a known limitation we do not claim to have solved. The paper's contribution lies entirely to the left of that wall.

Frictions: where C++23 and IEEE numerics break the chain. The axioms cannot always be discharged. The friction points encountered while building the library are observational, not advocacy, and each is pinned at a concrete file in source (with the relevant lines flagged in the corresponding partition's witness blocks) so a reader can verify the mechanism rather than take the claim on faith. *Integer-promotion shape mismatch*: `bool + bool` and `bool * bool` return `int` rather than `bool`; `unsigned short + unsigned short` / `unsigned short * unsigned short` and `unsigned char + unsigned char` / `unsigned char * unsigned char` likewise promote to `int`. The functor surface (`std::plus<bool>`, `std::multiplies<unsigned short>`) returns the carrier by signature, but the literal `+` / `*` operator surface does not (witnessed at `algebra/ring.cppm`, `algebra/field.cppm`, `morphologies/uint.cppm`, `order/lattice.cppm`). *Signed-overflow UB defeats $\mathbb{Z}$*: `int` / `long` / `long long` carry the literal ring operator surface (`HasRingOperators<int>` fires) but *not* the axiomatic ring witness, because closure-under-arithmetic fails as a stable axiom under signed-overflow UB; the negative pins `!IsArithmeticAdditiveGroup<int>` and `!category::IsRing<int, std::plus, std::multiplies>` at `morphologies/sint.cppm` make the rejection mechanical. *IEEE 754 breaks reflexivity*: `NaN <= NaN` evaluates to `false`, so `is_reflexive_v<double, std::less_equal<>>` is deliberately not registered, and `!IsTotallyOrdered<double>`, `!IsTotallyOrdered<long double>`, and `!category::IsRing<double, std::plus, std::multiplies>` all hold (`morphologies/floating_point.cppm` + `order/total.cppm`); the operational ordered-field reading survives via `IsOrderedField<double>` in `morphologies/archimedean.cppm`. *`std::variant` NTTP-unfriendliness*: the variant $\mathbb{Z}$-proxy `SignedCardinality` (variant of `SignedExtensionalCardinal<>`, $\pm\aleph_0$, `NaZ`) cannot be a non-type template parameter without workarounds, blocking compile-time arithmetic on the saturating integer carrier (`sets/cardinality.cppm`, `numbers/rational.cppm`). *`std::integral` unifies three structurally distinct algebras*: `std::integral T = std::unsigned_integral T || std::signed_integral T || std::same_as<T, bool>`, and the three siblings are a finite cyclic ring, an operator-surface-without-axioms carrier, and (separately) the Boolean rig + Galois field $\mathbb{F}_2$; nothing axiomatic survives the union (`morphologies/integral.cppm`). Each friction is the structural reason an axiom slot is left *unsatisfied* for the relevant family.

The policy gate as the bridge. Where the strict axioms cannot be discharged but the operator surface must still be programmed against, the library exposes an operational policy

[12] Worked diagnostic: a rank-deficient `Invertible2x2<Rat, 1, 2, 2, 4>` input produces a single named message (`Invertible2x2 requires full rank: a*d - b*c != 0`) at the `static_assert` site with one `note` pointing at the instantiation, no substitution-failure cascade. The predicate sits inside the class body as a literal `static_assert` rather than as a `requires`-clause to be probed, so the diagnostic fires at the named axiom directly.

gate. The canonical example is the rational-field claim, which the showcases program against without committing the carrier to a specific machine integer:

```
// numbers/rational.cppm --- operational field-like witness for ℚ.
// HasFieldOperators is the operator-surface gate; the strict
// axiomatic field witness lives at category::IsField but is not
// load-bearing for downstream code that only needs +, -, *, /.
static_assert(
    dedekind::algebra::HasFieldOperators<Rational<default_integer>>,
    "Rational<I> must satisfy the operational field-like witness "
    "(ℚ is a field).");

static_assert(
    dedekind::algebra::HasFieldOperators<
        Rational<SignedExtensionalCardinal<>>>,
    "Rational<SignedExtensionalCardinal<>> must satisfy "
    "HasFieldOperators: the fixed-width "
    "signed-rational carrier for ℚ.");
```

The two carriers differ in their machine-level friction profile (`default_integer` trips signed-overflow UB on extreme inputs; `SignedExtensionalCardinal<>` does not, at the cost of variant-NTTP friction) but both discharge the operational gate. Downstream combinators bind to `HasFieldOperators` rather than to `category::IsField`, and the carrier swap between the two is a one-line change at the `using` declaration. The gate is the load-bearing concept; the strict axiomatic field is reachable, on demand, when a use site warrants the heavier obligation.

In short, where Section G names the *mechanical* boundaries of the technique (template depth, `consteval` budgets, combinatorial blowup, diagnostics), this subsection names the *algebraic* boundary: which axioms discharge cleanly on which carriers, where the obligation falls to the engineer, and which frictions block the discharge. The empirical artefacts of the showcases sit one level below this chain: they demonstrate that the library realises the axiom → theorem discipline in C++23 today, on the carriers in the Rosetta tables of Section E.

G The Compiler as Structural Gate and Optimization Engine

Two durable uses of a compile-time type system underlie this paradigm, neither of which requires the compiler to be a theorem prover:

1. **Ruling out entire classes of errors.** Encoding algebraic laws as concept requirements makes every instantiation of a generic template an implicit proof obligation discharged at translation time. Failed obligations are type errors, not runtime exceptions. The obligation itself is finite, decidable, and mechanical: the compiler *checks* it, but does not *search for* it. Failed obligations surface as named-axiom messages (e.g. `Invertible2x2 requires full rank: a*d - b*c != 0`) rather than as deep template-cascade traces; the structural discipline behind this and a worked diagnostic are exhibited in Section G.

2. **Enabling entire classes of optimizations.** The LLVM backend exploits the structural knowledge inherited from discharged obligations. When a set is defined by a predicate and two predicates are logically contradictory, the compiler determines at translation time that the resulting intersection is empty and emits no machine instructions for it. This is what we term *structural pruning*, and it is the subject of Section 3.3.

Both uses are bounded but open-ended. clang's type checker is not a theorem prover: it has no dependent types, no universal-quantification inference, no proof-search engine. What it does have is `static_assert`, `requires`-clauses, and `constexpr` evaluation, enough to verify a predetermined proof sketch, not enough to generate one. The paradigm claims the reliable half: ruling out, and optimizing under, the classes of structural failures the programmer has named in the type system. The Curry–Howard correspondence between propositions and types is suggestive here, but its full force (dependent products, inductive proofs, universe hierarchies) is not available in C++23; what *is* available is enough to anchor the two uses above. Each use ruled out exposes a next class that the current machinery does not handle but the library's structural vocabulary already names; the four axes of Section F fall out of the halfspace reductions almost by construction.[13]

Three honesty-anchoring constraints. Three compounding constraints keep the library's artefacts honest: *domain textbook anchoring* (concept names and law statements taken directly from the published mathematical vocabulary: Lawvere, Mac Lane, Lambek, Wadler, Moggi); *mechanical validation* (every algebraic claim becomes a `static_assert` the compiler discharges; every concept composes through a named-module DAG; every behavioural claim passes CI); and *codomain textbook anchoring* (concepts validated against the C++ standard library's own ontology: `std::ranges::input_range`, `std::regular`, `std::numeric_limits`, the iterator hierarchy).[14]

A successful type-check is thus a *weak but universal* proof about the overall health of the Ontology: it asserts that every type-shaped claim in the build graph composes without contradiction, even though the individual assertions are coarse (concept satisfaction, not law-by-law verification). The automated test suite is the converse, a growing set of *strong but existential* proofs about the same Ontology: each test pins a concrete carrier-instance against a concrete law and discharges the obligation as a `static_assert` witness rather than a runtime equality

[13] The introduction's Gödel reading applies here: a formal system closing at one level motivates the next, not as exhaustion but as successor. The closest formal analogue is Turing's ordinal logics [58], where each stage's consistency feeds the next stage's formation rule; the operational reading is benign, with the development cycle of Figure 2 playing the successor role.

[14] The library is developed with AI assistance (Anthropic's Claude as the primary code-writing agent; Gemini and GitHub Copilot in supporting roles); the human role is prioritisation, orchestration, and consensus-before-merge. The three constraints above are what keep the workflow honest: the mathematical content stays in mathematical vocabulary all the way down to where the machine takes over.

check. Universal coverage is the build's job; existential depth is the test suite's; both accumulate monotonically under the convergence discipline. As a convenient and notable side effect: source code listings presented in this document are lifted (occasionally abridged for paper-page budget) from the repository's test suite; each listing's source-tree path is recorded as a LaTeX-source comment immediately above the listing, so a reader auditing against the repository can find the verbatim original directly.

Similarly, where the output of LLVM optimizations is reported here, these are examples which are lifted from the test suite. Here, the compiler is run during CI with appropriate flags to emit a portable representation of the optimized backend instructions. The emitted instructions are then compared to LLVM IR fixtures checked into the source tree. This way, it can be checked during CI that algebraic compile-time optimizations are recognized by the compiler during the development cycle. Regressions would lead to build failures and be flagged during CI as blocking the corresponding PR.

H Use of Agentic Systems and AI-Assistance Disclosure

Agentic systems (LLMs invoked as code-review and refactoring assistants) enter this picture as a third party that can act on either side of the universal/existential split, provided the source carries enough domain context to prime them on each invocation. The library therefore embeds the domain language deliberately deep into the source: exhaustive Doxygen on every public surface, opinionated quotes from practitioners (e.g. references to the Polish school of logic and semantic epistemology [59] where the underlying register is at issue), and named provenance on each trait specialisation. The audience is not only the human reader; it is also the agentic context window on every pull-request review. Prompts to such systems are accordingly written liberally and in *loose language*, in the careful, sparing register of RFC 2119 §6 [60] (where the normative weight of MUST / SHOULD / MAY is reserved for the cases that actually demand it), so that the agent can catalyse convergence not only at the syntactic layer the conventional build pipeline polices, but also at the semantic layer where loose intent has to be translated into precise concept-discharge obligations.

The concrete record of that assistance, and the disclosure the ‹Programming› journal's AI-tools policy requires, follows.

The development of `dedekind` relied substantively on AI assistance throughout, and the writing of this paper did so as well. The disclosure here is the concrete instance of the cost-benefit-calculus claim in §1: that paragraph names the shift in the abstract; this one names how it shifted on this artefact. Per the ‹Programming› journal's policy on AI tools [61], the use of these tools is acknowledged and disclosed; the AI systems named below are tools whose contribution is recorded here as assistance, not authorship. The author is a single human person, named in the byline, who bears full responsibility for content, scope, correctness, and editorial judgement.

The primary assistant was Anthropic's *Claude* family (Sonnet and Opus across the project's lifetime; the 1M-context Opus 4.8 variant for the most recent slices, where reproducibility against extended-context interactions matters), invoked through the Claude Code CLI. Google's Gemini was used for second-opinion structural critique on substantive mathematical claims (size-restriction notation, adjunction-encoding feasibility, foundational-vs-mechanical disentanglement). GitHub Copilot's pull-request reviewer was invoked on every PR for scope-vs-claim consistency checks at the prose / table / listing level. The assistance covered draft generation (prose, code, listings, tabular structured elements), structural refactoring (concept naming, partition placement, hub-and-spoke dispatch shape), test scaffolding, review-thread response drafting, and CI / GitHub workflow scaffolding. These tasks span the manuscript throughout, from its prose and listings to its tables and figures, rather than being confined to any single section.

The pattern is visible at commit-level granularity in the public repository at `https://github.com/vincentk/dedekind`: every commit message, every pull-request review thread, and the public CI build log compose into a complete audit trail. Reviewers can inspect the assistance pattern at any commit. "Assisted" rather than "co-authored" is the operative voice throughout, both for this paper and for the codebase it documents.