The Juliet Posture

Structurally Typed Functional Programming in Standard C++23

Vincent Kräutler[a]

a Unaffiliated

Abstract Parametric polymorphism, type classes and structural recursion are the tools with which a functional programmer says what a program does and has a type checker hold it to that. C++23, which runs much of machine learning and most embedded devices, has the material for them: compile-time evaluation of ordinary functions and of most of the standard library, concepts, and folds over ranges. It does not, however, hold a generic function to its declared constraints: the body may use any operation its type argument happens to have.

This paper characterises a fragment of C++23 as it ships, with no language modification, compiler extension or tool to install: an embedded language whose domain is System F and whose programs are ordinary C++23 programs. The fragment is type-class-style code under three rules, which we call the Juliet Posture (Jlt): i) values are copyable data and functions are pure and evaluable at compile time; ii) generic code varies with a type only through that type's concepts and traits; iii) recursion is structural, on a type or on a natural number that counts down. Its kernel, Featherweight Jlt (FJlt), translates into System F by dictionary passing, as Haskell's type classes do, and the translation is total and preserves types and evaluation, so an FJlt program inherits termination, parametricity and a value. What Jlt writes beyond the kernel is sugar, named with its desugaring, and that the desugaring keeps meaning is the paper's working hypothesis. Declared laws license overloads, and overload resolution composes them: in one program the action law and associativity, neither aware of the other, turn a fold of n steps into one of $\log n$, even at compile time, where no optimiser runs. Along the way the paper shows that C++20 concepts with overlapping overloads resolve incoherently in GHC's sense, and adopts GHC's rule. The evidence is proof sketches, listings that compile and run at both binding times under a test in the public repository, their compiled output, and a library whose core a linter checks against the rules.

The claim is one of existence: C++23 already contains an embedding of a fragment of System F, held by the compiler where the standard checks and by the programmer's discipline elsewhere, and available today wherever a stock C++23 compiler is.

ACM CCS 2012

- **Software and its engineering** → **Polymorphism**; *Functional languages*; *Formal language definitions*;
- **Theory of computation** → **Type theory**;

Keywords C++ concepts, type classes, System F, parametricity, structural recursion

The Art, Science, and Engineering of Programming

Perspective The Theoretical Science of Programming

Area of Submission Systems programming, Type systems, Generic programming



© Vincent Kräutler
This work is licensed under a "CC BY 4.0" license
Submitted to *The Art, Science, and Engineering of Programming*.

1 Introduction

The case for functional programming has been made the same way for decades, and it is a good case; even a compiler's own intermediate form, static single assignment, is a functional language [4]. A function that depends on its arguments alone can be replaced by its value: programs can be read as equations, rewritten by them, and reasoned about by them [6]. A function that is polymorphic in a type cannot inspect that type, so its type alone says a great deal about what it does [112]. A class of types with operations and laws lets generic code be written once against the operations and reasoned about by the laws [113]. The two are one mechanism with two sources of equations: a free theorem is proved from the type alone, for every term, and a law is declared of an instance. Either becomes a rewrite by Leibniz's law, which Rule 1 conserves (Section 2). For the declared laws Birkhoff's theorem makes the reasoning complete: an equation follows from the laws by equational reasoning exactly when it holds in every model of them [9, 15]. A class with laws is an equational theory and its instances the models, a variety, and a stronger class has fewer instances, so the constraints describe the structure and the structure licenses the optimisation. The goal is correctness by construction in Dijkstra's sense [28].

The case is usually made in a language built around it. This paper makes it in a mainstream systems language instead, C++23. Much of machine learning and most embedded devices run on C++, and a growing number of the people who write it arrive from Python with a functional vocabulary. For them the repository binds the library's combinators into Python as handles to the compiled core, so the laws declared in C++ hold in an interactive session: the specialisations they license run there at run time, and only what the compiler evaluates of them is fixed when the bindings are built (Section 6). The most widely written functional language may well be SQL's `SELECT`, a declarative query over relations [46], whose algebra is Codd's [21], with Tarski's calculus of relations [106] its older relative. A functional fragment of C++23 is worth drawing because the whole is too large to work in with confidence, and worth drawing now because C++23 is the first standard with every ingredient: compile-time evaluation of ordinary functions and of most of the standard library, concepts, and folds over ranges. Performance is why one comes to such a language in the first place, by its zero-overhead principle [96],[1] and the posture does not affect it: an optimising compiler already reasons about a program, by abstract interpretation [23] and by algebraic laws it is told it may assume, as `-ffast-math` lets it for floating point. What the posture addresses is the other cost, the language's size. Alignment with the posture is easier to check than a decade ago: for example, a language model of code [19] reads every change against its rules. And while this is not a hard proof but only a read, it is often a thorough read (Section 7).

[1] Stroustrup has since called the phrase an understatement: "It's compiled away. This is why I talk about zero overhead abstraction, and people are beginning to take me to task for that because that's understating and underestimating the ability of the C++ compiler. We can do negative overhead abstraction" [97]. Listing 5 is a case: a count over all sequences compiles away to a constant, a matcher's construction to its table.

C++ is not a theorem prover, but its compiler performs checks of the same kind: a compile-time check is a claim the compiler verifies, and the worth of such a checker when the author is not the one trusted had a recent highlight, machine-generated proofs [41] in Lean [75] whose correctness rests on the checker, not the author, although even such a checker can have loopholes [74]. The search that found those proofs is a black box, and a language model's account of its own reasoning need not be the reasoning it did: its chain of thought can leave out what drove the answer, and circuit tracing finds computations its explanation misdescribes [65, 111]. What makes the output usable is that it is no black box but an artefact a small checker verifies, an interpretable model in place of an explained one, and a registry of Lean-verified mathematics is arranged that way: a short statement a human reads, checked against a proof of any length, which a machine may find correct and still unreadable [89, 105]. A program is such an artefact too: a machine runs an executable with no interpretation, so every claim about it is a claim about something formal, regardless of whether that actually is what its author intended. For generic code, C++ lacks just such a check. **concept**s as proposed for C++0x checked a template's definition against its constraints [30, 39], but as standardised in C++20 they check only its uses [99], so a template's body may use operations its constraints do not name. The designers named the problem at the outset, in 2003: "crucial aspects of the programmer's understanding of a template cannot be expressed directly in the code for compilers and other tools to use" [100]. C++23 therefore contains the material for a typed functional fragment but does not enforce its boundary, and this paper documents a discipline that does, recovering some of what a functional language guarantees at the cost of three rules. The cost is paid once, in how a program is written, and what it buys is paid back at every use: a law holds in every model of it, a specialisation it licenses pays on every call, and a checked statement serves every reader.

The fragment is the *Juliet Posture*, or Jlt, and it has two boundaries, both drawn here. The outer boundary is drawn by the three rules: Jlt is a proper subset of C++23 defined by these three rules. We call it a posture because it asks only for a style of programming: every ingredient is in the ISO standard [47], implemented by toolchains in wide use, and what Jlt adds is the programmer's discipline. The compiler holds some of the rules and the programmer the rest, and a linter helps but is not required (Table 1). The inner boundary is drawn by the translation: the *kernel*, Featherweight Jlt (FJlt), is the part of Jlt on which a translation to System F ($[\![\cdot]\!]$) is defined, and it inherits from System F termination, parametricity and a value. Between the two lies *sugar*, in Landin's word [61], which this paper names and characterises without proof (Appendix B), the arrangement being the Haskell Report's, a kernel and the translations into it [66].

Jlt is thus an embedded domain-specific language in Hudak's sense [42], whose domain is System F and whose host is C++23. Its programs are ordinary C++23 programs, type-checked and compiled by an unmodified toolchain, Clang 23 in this case (Section 7). Being a strict subset of C++23, it is not a language this paper designs and develops as an architect might develop a house. Instead it characterises a fragment of a pre-existing ecosystem, as a traveller might discover a city. The claim here is one of existence: C++23 contains an embedding of a sizeable fragment of System F, and

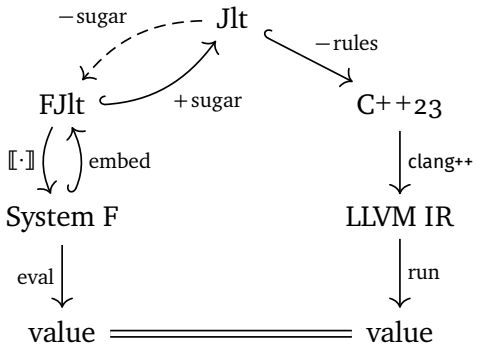


Figure 1 The arrows of the paper. The inclusions of FJlt in Jlt and of Jlt in C++23 are proper, and each is labelled with what it adds: Jlt is the kernel and the sugar, C++23 is Jlt without the rules. The translation $[\![\cdot]\!]$ is total on FJlt (Section 4), and *embed*, defined on its image, a non-empty part of System F, is a section of it [77]. Its composite with the two inclusions is a partial map from System F, the embedding that makes Jlt an embedded language: for every closed t of base type in the image, $\mathit{eval}\, t = \mathit{run}\,(\mathsf{clang{+}{+}}\,(\mathit{embed}\, t))$, by Theorem 2 inside the calculus and the argument of Section 6 for the program the compiler emits. clang++ is defined on every C++23 program within the implementation's limits (Sections 6 and 7). Solid arrows are established: for a kernel program the two readings agree on the value, by Theorem 2 inside the calculus (Section 5) and by an argument for the program the compiler emits (Section 6, Appendix G). The dashed arrow is the working hypothesis of Section 3: *desugar* is total on Jlt and a retraction of the inclusion, $\mathit{desugar} \circ \mathit{incl} = \mathit{id}$ exactly, by construction, and $\mathit{incl} \circ \mathit{desugar} = \mathit{id}$ up to the value a program computes, so that the bottom edge holds for every Jlt program p, $\mathit{eval}\,[\![\mathit{desugar}\, p]\!] = \mathit{run}\,(\mathsf{clang{+}{+}}\, p)$.

the listings exhibit it. No claim is made that Jlt is unique or canonical, although it seems likely that another fragment drawn to the same ends would look much like it. As in the tagless-final style of Carette, Kiselyov and Shan [18], the embedded terms are generic code over an interface, here a **concept**, and their interpretations are its models. The posture's name is a nod to structural typing [17, 80]: in C++23 a **concept** is a predicate on a type's structure, not its lineage, and so a rose by any other name would smell as sweet [91]. Figure 1 sketches the arrows between these pieces.

Contributions. The contribution is a map from a known discipline onto existing mainstream functionality, not a piece of software.

1. The posture: three rules on standard C++23, each with what the compiler checks of it and what it leaves to the author (Section 2).
2. Featherweight Jlt, the kernel: a core calculus whose syntax is a subset of C++23 as written, with typing, evaluation and a translation into System F. Three theorems about the translation and three corollaries, with proof sketches (Section 5, Appendix F).
3. An adequacy argument from C++ to the calculus, which states its gaps rather than writing around them (Sections 3 and 7), and the sugar, named with its desugaring (Appendix B).

4. An observation about C++20 concepts: with overlapping constrained overloads, a generic function's behaviour can depend on a constraint it never declared, which is GHC's incoherent instance resolution, by default and silently. Jlt adopts GHC's rule, and the repository's linter reports a call at a value of parameter type that breaks it (Appendix C, Section 7).
5. The exhibits: every listing in this paper compiles and runs under a stock compiler at both binding times, by a test in the accompanying repository [57], and is held to the three rules by a mandatory review against them and by a mechanical linter in the same repository: that is what compliance means here.

Not contributed: the translation, which is Wadler and Blott's [113]; the calculus-with-concepts, which is Siek and Lumsdaine's F^G [92]; and complete definition checking: Section 7 says what the repository's linter reads and what it misses.

How to read this paper. The paper is written for authors of generic C++ libraries first, with System F as the warranty and a rule for their overloads; for those who audit C++ against a policy and need to know what a tool checks; for readers of programming languages in the line of concepts and type classes; and for researchers who arrive from Python and want the performance of C++ with a guard rail. The sections follow the arrows of Figure 1, one each. Section 2 is the right hook, the rules that carve Jlt out of C++23; Section 3 is the left hook, the sugar: which Jlt programs are kernel programs; Section 4 is $[\![\cdot]\!]$, the translation of Featherweight Jlt into System F; Section 5 is *eval*, the theorems read back from System F; Section 6 is `clang++` and *run*, one program at both binding times, and how instantiation relates to the translation. The listings follow one example through the paper, and Figure 2 shows where its structures live: `concat`, a monoid at the lengths that need no recursion, then the fold over the chain of a count under an arrow, then Fibonacci's matrix chain, which two declared laws fold in logarithmically many steps, then restriction sites, where the compiler partially evaluates a matcher in its pattern. Section 7 says where the arrows are partial, Section 8 where the work sits, and the calculus is in the appendices.

2 From C++ to Jlt

Jlt is standard C++23 under three rules, not a dialect and not a library; the compiler enforces some of the rules, and we say which. Each rule has a listing of a few lines, verbatim from the repository's test up to formatting and the definitions a listing omits or the test imports; the LLVM IR of Appendix G is Clang 23's output, its flags in its caption, with names shortened.

Rule 1, Substitution: values and pure functions. Every type a Jlt program passes around is `std::semiregular`: default constructible, copyable, and nothing else is assumed. Every

function is **constexpr** and depends on its arguments alone.[2] What the rule conserves is what has come to be called Leibniz's law, that equals cannot be told apart by any operation, at either binding time, whose converse is the principle Leibniz himself stated [63, 64]: a call may be replaced by its value, $e[f(x)] = e[v]$ whenever $f(x)$ evaluates to v. In a programming language the law is referential transparency [95], and a language where $a = b$ but $f(a) \neq f(b)$ for some f has an equality one cannot reason with.

In Listing 1, **const** T and **const** T& are one parameter, the second an optimisation the translation erases, since nothing changes the value the reference denotes. An arrow is a function made a value, and its laws are equations between values, so == is where Leibniz's law meets the program: the kernel asks it of no type, the listings of every carrier a law is stated on. The properties a law singles out, mono, epi and iso by cancellation [79], involution, the variances, are certificates declared or derived on the arrow, whose type names carrier and operation, so a certificate is about the signature. Two consequences keep that exact: a function returns a value, never a reference, and every member function, a call operator included, is **const**. The grammar of Appendix A has neither a reference type nor a non-**const** member, and the linter of Section 7 holds C++ to the same. The compiler enforces the rule in part: a **constexpr** function that reaches for state fails to evaluate at compile time, but nothing stops a caller from only ever calling it at runtime.

Rule 2, Abstraction: a type varies behaviour only through its concepts and traits. Generic code may do with a T what T's concepts and traits say it can, and nothing else. That is the premise of Wadler's free theorems [112], with concepts and traits in the place of type classes [113], and what it conserves is the body's independence from the carrier: f<T> is one generic term applied to T's dictionary, so it respects every relation the dictionary respects (Corollary 2).

concat uses only what its constraint names, so it is a function of the monoid's record. A C++ compiler checks a template's body at instantiation, against the type supplied, and not at definition, against the constraint, which is why this is a rule here and not a theorem.

Rule 3, Descent: recursion is structural. A Jlt program recurses on a structure whose residue, what remains to be folded, gets smaller at every step and so reaches its end. The step is Figure 2's *prior*, which steps down and has nothing at the bottom, or another step down the same order, as halving is (Listing 4), on an order with no infinite descent. What the rule conserves is that inductive argument: structural induction [16], a language all of whose programs end by it being, in a stricter form, Turner's total functional programming [110], and the fold's universal property its

[2] A **constexpr** function may still change its own locals: constant evaluation permits modifying only objects whose lifetime began within it, so no mutation can leak, which is the encapsulated state of runST [62]. At runtime the linter of Section 7 holds the body to the same condition by closing each channel a mutation could leak through: parameters, results, the object, statics, globals, **volatile**, boxes, and escaping closures that capture by reference.

```cpp
// Identity arrows:
template <typename A>
constexpr auto id() { return Identity<A>{}; }
constexpr auto id_b = id<bool>();
constexpr auto id_u = id<size_t>();
static_assert(
id_b(true) == true && id_u(42) == 42 &&
is_monotone(id_u) && is_monotone(id_b) &&
is_involution(id_u) && is_involution(id_b) &&
is_identity(id_u » id_u) &&
(id_u » id_u)(42) == 42 &&
is_monotone(id_u » id_u) &&
is_involution(id_u » id_u));
```

```cpp
// Constant arrows:
constexpr auto nil = constant<bool>(0uz);
constexpr auto one = constant<bool>(1uz);
constexpr auto ans = constant<bool>(42uz);
constexpr auto aye = constant<size_t>(true);
constexpr auto nay = constant<size_t>(false);
static_assert(
(ans » id_u)(true) == ans(false) &&
is_constant(ans) && is_monotone(ans) &&
is_constant(aye) && is_antitone(aye) &&
!is_monic(ans) && !is_epic(ans) &&
!is_monic(aye » id_b) &&
!is_epic(nay » id_b) &&
is_constant(nil » nay » ans));
```

```cpp
// Involutions, pointwise:
static_assert(
(!aye)(0) == false &&
is_involution(!id_b) &&
is_constant(!aye) &&

(~nil)(true) + 1 == 0 &&
is_involution(~id_u) &&
is_constant(~nil) &&

(-one)(true) == (~nil)(true) &&
is_involution(-id_u) &&
is_constant(-nil) &&

is_iso(-id_u » ~id_u) &&
!is_involution(-id_u » ~id_u));
```

```cpp
// Products and coproducts:
constexpr auto fst = Π<1, pair<bool, size_t»{};
constexpr auto sec = Pairing{id_b, nil};
constexpr auto inl = ι<1, bool, size_t>{};
constexpr auto ret = Copairing{id_b, nay};
static_assert(
is_epic(fst) && !is_monic(fst) &&
is_monic(sec) && !is_epic(sec) &&
(sec » fst)(true) && !(sec » fst)(false) &&
is_monic(inl) && !is_epic(inl) &&
is_epic(ret) && !is_monic(ret) &&
(inl » ret)(true) && !(inl » ret)(false));
```

Listing 1 Rule 1: Substitution. Equals cannot be told apart, so a function is its graph, and functions are the arrows of a category, with composition » their only structure. Top left: the identities. Top right: the points, constant arrows, monotone and antitone at once and neither monic nor epic. Bottom left: the involutions, each the C++ operator post-composed, so !f is $\neg \circ f$; two compose to an isomorphism and to no involution. Bottom right: a projection and its section, an injection and its retraction, the arrows that are epic or monic but not both. A law is asked on the arrow's value, at compile time: is_monic(f) reads what the concept IsMonicArrow reads, and a composite's law is derived from its legs.

```cpp
using namespace std;

template <typename T, typename Op>
constexpr T unit(Op) = delete;
template <typename T, typename Op>
constexpr bool associative(Op) = delete;

template <unsigned_integral T>
constexpr T unit(plus<T>) { return 0; }
template <unsigned_integral T>
constexpr bool associative(plus<T>) {
 return true;
}
template <typename T, typename Op>
concept HasUnit = requires(Op op) {
 { unit<T>(op) } -> same_as<T>;
};
template <typename T, typename Op>
 requires HasUnit<T, Op> &&
 ↪  equality_comparable<T>
constexpr bool is_unit(const T& a, Op op) {
 return a == unit<T>(op);
}

template <typename T, typename Op>
concept IsMonoid =
   semiregular<T> &&
   requires(const T a, const T b) {
    { Op{}(a, b) } -> same_as<T>;
    { unit<T>(Op{}) } -> same_as<T>;
    { is_unit(a, Op{}) } -> same_as<bool>;
   } && associative<T>(Op{});
```

```cpp
template <typename T, typename Op = plus<T>»
 requires IsMonoid<T, Op>
constexpr T concat() { return unit<T>(Op{}); }
template <typename T, typename Op = plus<T>»
 requires IsMonoid<T, Op>
constexpr T concat(T w) { return w; }
template <typename T, typename Op = plus<T>»
 requires IsMonoid<T, Op>
constexpr T concat(T v, T w) { return Op{}(v, w); }

static_assert(
concat<size_t>() == 0 &&
concat<size_t>(42uz) == 42 &&
concat<size_t>(40uz, 2uz) == 42 &&
concat<size_t, multiplies<size_t»(6uz, 7uz)
   == 42 &&
concat<string>() == "" &&
concat<string>("ab"s, "c"s) == "abc" &&
concat<bool, logical_or<bool»(false, true) &&
concat<bool, logical_and<bool»(true, true) &&
concat<Sections<size_t>, compose>()(42uz)
   == 42 &&
concat<Sections<size_t>, compose>(
   adding(40uz), adding(2uz))(0uz) == 42);
```

Listing 2 Rule 2: Abstraction. Left: a monoid is a set which is closed under an associative binary operation and equipped with a unit element. Its elements need not be comparable, since equality is not decidable on every carrier, on functions for one. The unit is identified instead: `is_unit`, by default equality with the unit, which a carrier may supply otherwise. The monoid shape is certified for the family of `std::unsigned_integral` under (`0`, `+`). Right: `concat` at the lengths that need no recursion, none, one and two. The monoid signatures are: **`size_t`** under (`0`, `+`), (`1`, `*`), **`bool`** under (`false`, `||`), (`true`, `&&`), `string` under (`""`, `+`), and `Sections`, the sections of `+`, under (`id`, `»`), with `adding` the curried `+`.

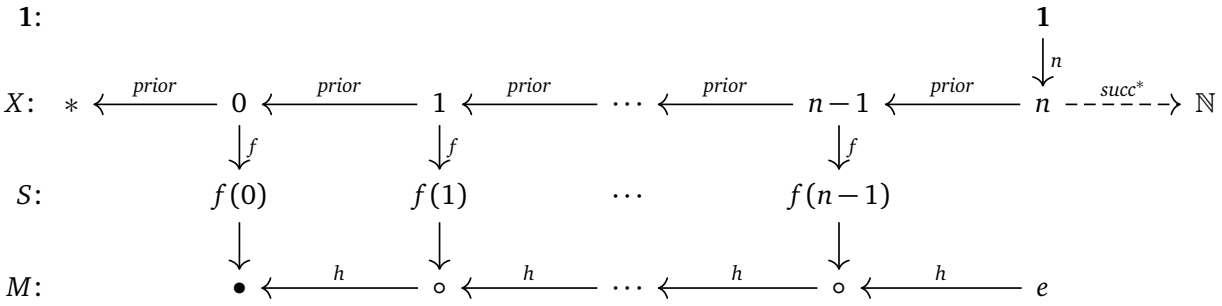


Figure 2 Where each structure lives. Seed: after Lambek [60], the structure map $[0, succ] : \mathbf{1} + X \to X$ of the naturals is an isomorphism. Its inverse $prior : X \to \mathbf{1} + X$ steps down and always ends. Peano's *succ* [78], dashed, never does. The user supplies the point $n : \mathbf{1} \to X$ and *prior*. The chain is what *prior* reaches from n: the n elements below it, not n itself. So n names the chain's cardinality, and 0 the empty chain. Order: after Dedekind [27], a chain is a set closed under a map into itself. The chain of an element is its forward orbit, here the chain of $prior\, n$. The carrier needs neither an order nor an origin. Reachability under *prior* orders the chain, and its bottom is where *prior* has nothing. Algebra: f lands in S, and the fold in M. In the simplest case $S = M$ is a monoid. A monoid S may also act on M, and a semiring on a semimodule, which adds distributivity. Fold: a list has two folds [36]. The one along the walk, $h : M \times S \to M$, runs from e at the seed's end through the intermediate values, ∘, to the result, •. It needs no list: it is a stream and its iterator [94]. With *succ* as well, $succ >> prior = \eta$, the chain is a zipper [43]. With offsets it is an array [54, §2.2.2]. These are the standard's forward, bidirectional and random-access iterators [47].

equation [44, 69], $h = fold\, a$ exactly when $h \circ in = a \circ F\, h$.[3] Meijer et al. take their recursion operators in CPO, where one list type holds the finite, the infinite and the partial lists, and a fold over it need not end; Jlt takes them for data only, the initial algebras in Set, where a list in a type is finite as an initial algebra and the chain in a value is finite by the descent, and codata is out of scope: a coalgebra is admitted as a generator bounded by a count, never as a value. The structure lives in a *type* (a typed term, an array of known length), or in a *value*. In both cases, an iteration that can be mapped onto the chain of Figure 2 contains no loop and terminates, whether the chain is held as one count or as the elements of an array. In the terms of Meijer et al., concat over a list is a catamorphism. The chain is an anamorphism, *prior* unfolded from the seed and delimited where it has nothing. concat over the chain is a hylomorphism, and the chain is never built as a list, only as its fold (Appendix D).

What the rule buys is termination and, with Rule 2, the System F reading, data as their own folds after Böhm and Berarducci [11] (Appendix D). Like System F,

[3] F is one layer of the data type with a hole for the rest, for a list of words $F\, X = 1 + Word \times X$; *in* builds one more layer, the constructors *nil* and *cons*; a, the algebra, says what to do with one layer once the rest is folded; and $F\, h$ applies h to the rest only. For concat the equation is its definition, $h(nil) = unit$ and $h(cons(w, r)) = w \oplus h(r)$, and *exactly when* says that concat is the only function that satisfies both.

```cpp
// the counts, a finite chain
static_assert(IsFinite<size_t> &&
   IsTotalOrder<size_t, less_equal<>>);
constexpr auto ns =                  // 9, ..., 0
   10uz » chain<size_t>();
static_assert(
   ranges::input_range<decltype(ns)> &&
   ranges::distance(ns) == 10 &&
   ranges::equal(ns, views::iota(0uz, 10uz)
                   | views::reverse) &&
   ranges::empty(0uz » chain<size_t>()) &&
   ranges::equal(ns » id<size_t>(), ns));
```

```cpp
// Collatz's step, until 1
constexpr auto prior = until(equals(1uz),
   arrow<size_t>([](size_t n) {
     return n % 2 ? 3 * n + 1 : n / 2;
   }));
constexpr auto cs =        // 500, 250, ..., 1
   1'000uz » chain(prior);
static_assert(
   ranges::input_range<decltype(cs)> &&
   ranges::distance(cs) == 111 &&
   ranges::max(cs) == 9232);
```

```cpp
// the fold on a monoid: the library's concat
template <ranges::input_range R,
          typename T = ranges::range_value_t<R>,
          typename Op = plus<T»
  requires IsMonoid<T, Op>
constexpr T concat(R r, Op op = {}) {
  return ranges::fold_left(r, unit<T>(op), op);
}
static_assert(
   concat(ns) == 45 &&
   concat(ns, multiplies<size_t>{}) == 0 &&
   concat(ns » constant<size_t>(7uz)) == 70 &&
   concat(cs) == 102512);
```

```cpp
// the fold on an action: matrices on vectors
using V = array<size_t, 2>;
struct M { V top, bottom;          // row by row
  friend constexpr V operator*(V v, M a) {  // v·A
    return {v[0] * a.top[0] + v[1] * a.bottom[0],
            v[0] * a.top[1] + v[1] * a.bottom[1]};}
  /* M * M, M + M, n * M, == */ };
constexpr auto act = [](V v, M a) { return v * a; };
constexpr M fib{{1, 1}, {1, 0}};
constexpr auto ms = ns » constant<size_t>(fib);
// F(11), F(10)
constexpr auto fs_10 = ranges::fold_left(
  ms, V{1, 0}, act);
static_assert(fs_10 == V{89, 55});
```

Listing 3 Rule 3: Descent, Figure 2 in code. Top. Left: counting down, a finite chain, whose type supplies *prior*. Right: a chaotic path, Collatz's step from 1000. It is finite by fact, not by Rule 3: that every natural reaches 1 is an open problem [59]. Bottom. Left: a closed operation, concat on any monoid, generically, Listing 2's at a range. Right: a general fold is a heterogeneous action, here matrices on vectors, Fibonacci's. In the standard's ranges library [47], Figure 2's chain is an input_range. The arrow f along it is views::transform, and the fold with the action h is ranges::fold_left.

strongly normalising and so unable to express every computable function [37], Rule 3 trades expressiveness for termination, since the halting programs are not a decidable set [109]: a bounded search ends without always deciding its question (Section 6).

rule	archetype	keeps out of C++	buys	costs
1 Substitution	id, the identity arrow	state, pointers, exceptions	one program, two binding times	no state beyond a call's own locals, no I/O
2 Abstraction	concat, a monoid's fold at a length in its type	a body using what its constraints do not name	the System F reading	no ad hoc use of a type
3 Descent	concat over a chain, a fold over structure in a value	unbounded loops, general recursion	termination	a third answer, *out of budget*

Table 1 The posture at a glance, each rule named for what it conserves, with what it keeps out of C++: the filtration from C++ to Jlt. At build time, Rule 1 is held by the compiler in part and by the repository's linter for its syntax; Rule 2 by the linter at the level of names and by an auditing reader for the rest; Rule 3 by the compiler at compile time and by the linter for loops and recursion. Section 7 says how.

3 From Jlt to Featherweight Jlt

This section is the —sugar arrow of Figure 1, between the rules of Section 2 and the kernel's grammar of Section 4: which construction of Jlt becomes which term of the kernel, each closure statement left to Appendix B. Desugaring rewrites a program into one that uses only the kernel's constructions and computes the same value, as the C++ standard defines a lambda as a class with a call operator; no tool runs it. Tables 6 to 8 state it construction by construction: a condition, and the kernel term it yields.

A branch on a type, the sugar that matters most, becomes a family of generic functions ordered by specificity (Appendix C), which Theorem 3 keeps coherent. Listing 4's two specialisations, each licensed by a law, compose under it. The other sugar is routine (Table 6).

The theorems of Section 5 are about FJlt, and they reach a C++ program on two conditions, which hold in different ways.

Every FJlt program is a Jlt program, with the same meaning. This direction holds by construction (Appendix A): the grammar of Figure 3 is C++ syntax, FJlt's evaluation is constant evaluation restricted to it, each construct keeps the standard's rule, and the standard library pieces the grammar admits keep their standard meaning. Constant evaluation refuses undefined behaviour, so at compile time an FJlt program cannot go wrong silently; at run time it can (Section 7).

A Jlt program written in the grammar of Figure 3 is an FJlt program. This direction is not a theorem, because the compiler does not check two of the rules. It is also conditional on the grammar, because C++ has constructions FJlt lacks, and a Jlt program may use them.

	Jlt may write	kept out of the kernel by	the kernel has instead
1	overloading	the grammar (Appendix A), except by constraints	generic functions of one name ordered by their constraints, plus their specialisations
2	argument-dependent lookup	the grammar, and the lookup wall (Section 7)	instances found by the trait's name
3	implicit conversions	the grammar, and Rule 2 for the body	a conversion only where a requirement declares one, and the body then sees the declared type
4	references, moves	erased by the grammar (Appendix A): an optimisation, not a meaning	values copied; `const T` and `const T&` one value; no reference returned
5	the sugar: branches on a trait or a concept, lambdas, inheritance, `*` on an optional, …	the grammar, and desugared (Tables 6 to 8)	`?:` on a law, specialisation, an operation struct, a registration, the case form on a sum, …

Table 2 The filtration from Jlt to its kernel FJlt. The filtration from C++ to Jlt is the rules' (Table 1).

No row of Tables 6 to 8 needs anything the calculus lacks, and by measurement they are complete for the `category` module and the listings, at the resolution of a syntax node's kind (Appendix B). That each desugaring keeps the meaning is named, not proved: it is the paper's working hypothesis.

4 From Featherweight Jlt to System F

This section is about one map, the translation $[\![\cdot]\!]$ from Featherweight Jlt into System F with records and the iterators of inductive types, defined in full in Appendix D. It is total on FJlt, and Section 5 says what the image guarantees; it is partial on C++, and Section 3 is the argument for when a C++ program is an FJlt program. Six of its rules carry the listings' structure, and Table 3 states them. Conditionals, conversions and the remaining operators are routine, and Appendix D gives them all.

A law is a field of type `Bool`, and a shape built from shapes is a record holding their records, which is how a superclass is passed. The whole is dictionary passing, Wadler and Blott's translation of type classes, here into System F, the record being the dictionary [113]. Table 4 names the C++ devices it meets. `Op` is a type with exactly one value, the closure `Op{}`, so under translation the type *is* that closure, and `Op{}(a, b)` is `d.op a b`.

FJlt	System F	in
concept C = Ā	$Dict_C\ \overline{\alpha} = \{\ \overline{r : [\![A]\!]}\ \}$	Listing 2
constexpr τ f($\bar{\tau}$) = delete	nothing	Listing 2
template <> constexpr τ f<$\bar{\tau}$>(x̄) { e }	$\lambda \overline{x}.\, [\![e]\!]$	Listing 2
template <ᾱ> requires Q̄ constexpr τ f(x̄) { e }	$\Lambda \overline{\alpha}.\, \lambda \overline{q : Dict_Q}.\, \lambda \overline{x}.\, [\![e]\!]$	Listing 2
f<$\bar{\tau}$>(ē)	$[\![f]\!]\ \overline{[\![\tau]\!]}\ \overline{d}\ \overline{[\![e]\!]}$	Listing 2
fold_left(r, a, f), r a chain	$\pi_2\,(iter \mid r \mid step\,([\![r]\!], [\![a]\!]))$, *step* the chain's *next* beside $[\![f]\!]$ at its *head*	Listing 3

Table 3 Six translation rules exercised by the listings; a chain is iterated for its length, its position carried beside the accumulator (Figure 2). The whole translation is Figures 6 to 8 in Appendix D.

C++23	type-class reading	under translation
concept	class	record type
requirement met by the type's own operation	an instance nobody wrote	field filled by a primitive
explicit specialisation of a deleted function template	instance	field supplied by a value
constrained overload	instance with a context	function from records to a record
requires clause on a function	class constraint	one more parameter

Table 4 Dictionary passing, met in C++. The second row has no counterpart in a nominal type-class system: concepts are structural, so a type that has the operation satisfies the shape without a declaration.

4.1 Instances with contexts

An instance may itself require instances: the unit of a pair is the pair of units, provided both components have one. Listing 2's instance for the unsigned integrals is the simplest case, one context on one carrier. In C++ that is a generic function with a **requires** clause, and the clause is the context. Under translation it is a function from the components' records to the composite's:

```
pair_unit  =  Λ A. Λ B. λ da : MonoidDict A OpA. λ db : MonoidDict B OpB.
          (da.unit, db.unit)
```

The compiler applies such functions outside in, as often as the composite nests. The recursion ends, because a context on a structured head mentions only proper parts of its type arguments, and a context mentions only shapes declared before it.[4]

[4] The termination conditions on instance contexts are studied by Sulzmann et al. [102]. GHC calls them the Paterson conditions. FJlt's well-formedness requires them (Appendix C).

5 From System F Back to Featherweight Jlt

The theorems rest on FJlt alone. Their proofs (Appendix F) are inductions over the grammar, the typing and the translation of FJlt, and never mention the three rules. The rules are not the hypotheses of the theorems. They are what makes a C++ program a term of the calculus the theorems are about, and that step is the argument of Section 3.

Theorem 1 (Type preservation)**.** *If* $\Delta; \Gamma \vdash e : \tau$ *in FJlt, then* $[\![\Delta]\!]; [\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau]\!]$ *in the target, System F with records and iterators (Appendix D).*

Each **requires** clause has become a record parameter. At each call, the instance the compiler resolved supplies a record of the type that parameter demands.

Theorem 2 (Simulation)**.** *If* $e \longrightarrow e'$ *in FJlt (Appendix E), then* $[\![e]\!] \longrightarrow^{+} [\![e']\!]$ *in the target, up to the reduction of a fold's count, a descending self-call's fuel and a node's reading (Appendix F), and a value of base type translates to a constant. For closed terms of base type,* e *evaluates to* v *if and only if* $[\![e]\!]$ *reduces to* $[\![v]\!]$*.*

What a **static_assert** sees is what the translation computes, and, by Rule 1, what the runtime computes. Section 6 is this theorem run twice. The theorem also fixes the notion of equality, which equational reasoning presupposes and which is a question of its own [68]: two closed terms of a carrier are equal when they evaluate to the same value, by behaviour and not by identity, and Leibniz's law of Rule 1 holds by substitution. That is why Rule 1 asks std::semiregular and not std::regular: no == is asked of a value, a carrier's == is one more operation of its signature where a shape asks for it, and a function object, a value under Rule 1, has none.

Theorem 3 (Coherence)**.** *The translation is a function of the program: two resolutions of the same constraint at the same concrete types yield the same record.*

In C++ this is overload resolution together with the one-definition rule: at concrete types a specialisation is unique, or the program is ill-formed. It gives "the unit of + at **size_t**" one meaning in a program, so a law can be stated about it, and it lets laws compose: two overloads written apart are chosen the same way wherever they are called, as the two behind Listing 4 are.

Corollary 1 (Termination)**.** *Every FJlt program terminates.*

It follows from Theorem 2 and strong normalisation of the target (Appendix D). Every step of the program is at least one step of its image, and the image cannot step forever. The count in Rule 3 is therefore not a guard against non-termination. It is how a bound is said.

Corollary 2 (Parametricity)**.** *A generic FJlt function satisfies the free theorem of its System F type.*

By Theorem 1 and Reynolds' abstraction theorem [88, 112]. concat at a monoid can only combine the words by op, starting from unit, which is the free theorem of a fold. It is also why an instance with a context (Section 4.1) can read a composite's law off its components' records. The composite cannot do anything the components' records do not provide.

```cpp
// the overload associativity licenses
template <typename T, typename Op,
          IsCountWord N>
  requires IsMonoid<T, Op>
constexpr T power(const T& x, N n, Op op) {
  if (n == 0) {
    return unit<T>(op);
  } else {  // the halves are equal: one, twice
    const T h = power(x, n / 2, op);
    const T hh = op(h, h);
    return n % 2 == 1 ? op(hh, x) : hh;
  }
}
template <ranges::input_range R,
          typename T = ranges::range_value_t<R>,
          typename Op = plus<T»
  requires IsMonoid<T, Op> && IsCopies<R>
constexpr T concat(R r, Op op = {}) {
  const auto n = copies_count(r);
  return n == 0 ? unit<T>(op)
                : power(copies_value(r), n, op);
}
```

```cpp
// the action law, declared
using Act = remove_cvref_t<decltype(act)>;
template <> struct acting_monoid<Act> {
  using type = multiplies<M>;
};
// the overload it licenses
template <ranges::input_range R,
          semiregular M, typename H>
  requires IsCopies<R> && IsRightAction<
             H, M, ranges::range_value_t<R»
constexpr M fold(R r, M v, H h) {
  using Op = acting_monoid<H>::type;
  return h(v, concat(r, Op{}));
}
```

```cpp
// Listing 3's fold, by the library: the laws choose how
constexpr size_t fibonacci(size_t n) {
  return fold(n » chain<size_t>() » constant<size_t>(fib), V{1, 0}, act)[1];
}
static_assert(fibonacci(10) == fs_10[1] &&
              fibonacci(100'000'000) == 14'139'011'350'745'967'675uz);
```

```
fibonacci = λ n. π₂ (fold act (1, 0) ms)      -- ms = map (const M) (chain prior n)
          = λ n. π₂ (act (1, 0) (concat (·) ms))      -- by the action law
          = λ n. π₂ (act (1, 0) (Mⁿ))                 -- by associativity, squaring
```

Listing 4 Laws composing, on Listing 3's chain. Top left: the monoid concatenation $M \cdot M \cdots M$ is replaced by the power M^n, implemented by repeated squaring [55, §4.6.3]. Top right: the type constraints license the power method (Rule 2 for the specialisation, Theorem 3 for the composition). Middle: the power method keeps the 10^8 terms of the Fibonacci sequence within the compile-time step limit (Table 5). Bottom: the System F translation closely resembles the C++ above (Figure 1).

Corollary 3 (Type soundness)**.** *A well-typed closed FJlt term of base type evaluates to a value.*

This is the theorem a reader of Featherweight Java [45] expects first, soundness, here stronger because every term ends, and cheap once the rest is in place. A closed well-typed term is a value or takes a step (progress, Lemma 4), and it cannot step forever (Corollary 1), so it reaches a value. What the translation adds is the second half of Theorem 2: the value reached is the value the System F image computes.

Certified and inferred laws. A program declares few laws and infers the rest. A declared law is an axiom, its certificate evaluated where a record is built. A law at a construction is derived from the laws of its parts, through an instance with a context, by the entailment of qualified types [48]: that a finite power of a semiring is a semimodule over it is inferred, not declared (Appendix C). The inference is local in Pierce and Turner's sense [81]: information moves only between adjacent nodes, and here only upward, from a term's immediate parts to the term, a synthesized attribute [53]. They infer type arguments and Jlt infers laws, so the likeness is in the flow of information, not in what is inferred. By Theorem 3 each inferred law is unique, and the conditions on contexts that end instance search [102] end the inference. As type checking rules out classes of errors without proving a program correct, law inference licenses classes of optimisations without deciding program equivalence, which stays undecidable where every program ends: a program testing whether a machine halts within n steps equals the constant false exactly when that machine never halts [109]. Each optimisation is local, a rewrite at the node whose law licenses it, and so mechanical; the global one, the best equivalent program, is the undecidable one. Since the laws are the application's, not the machine's, so are the optimisations they license.

6 Partial Evaluation at Build Time

Every function in this paper has run twice already. The test behind the listings calls the same functions at runtime, on values it did not have at compile time. Rule 1 is what makes that unremarkable: a **constexpr** function of its arguments alone has no binding time of its own.[5] Theorem 2 says the two evaluations agree, which is the presumption of this section: moving a call from compile time to run time changes where it is evaluated, not what it evaluates to. Both are partial evaluation [49] at build time: the compiler evaluates what it can and leaves the processor the rest. Listing 5 is the paper's exhibit on both of its panels, each followed through the translation. With the sequence free, the compiler evaluates what it can, the table, and keeps the fold over the sequence, which only the run time can finish. What remains is the loop the optimiser emits, read as a function after Appel [4]: at `-O2` it allocates nothing and calls nothing, and each letter is four loads, the letter, its column, the next state and

[5] A **consteval** function is evaluated only at compile time, so it has one binding time, which is why Rule 1 asks **constexpr** and not **consteval**.

whether that state accepts; each call first copies the table by value, as Rule 1 has it. With the sequence given and bound by **constexpr**, the term evaluates to its count, and so does the code: Figure 1's bottom edge on one program. Given but not bound, the optimiser keeps the loop, so the constant is the binding's and not the optimiser's.

The presumption holds where both evaluations finish and neither meets undefined behaviour. Table 5 lists where the two binding times part.

	at compile time, in constant evaluation	at run time
who evaluates	the front end's interpreter, one step at a time	the processor, after the optimiser
a fold's length	at most 1 048 576 steps (`-fconstexpr-steps`), past which the program does not compile	the count alone
a call's depth	at most 512 nested calls (`-fconstexpr-depth`)	the stack
undefined behaviour	refused: the expression is not a constant, and the program does not compile	no diagnostic required and no guarantee: the optimiser may assume it never happens
a shortcut	only by resolution, a declared law, as Listing 4's overloads	the optimiser's, as it fuses Listing 5's stages into one loop
telling them apart	`if consteval` and `is_constant_evaluated()` say compile time	they say run time, and FJlt desugars it to the first where both agree

Table 5 Where the two binding times differ, for Clang 23 with its default limits. Each limit counts the evaluator's work, and can be raised. Apart from the last row, no difference is a different value: where both evaluations finish without undefined behaviour, they agree (Theorem 2).

The two evaluators are different programs. In constant evaluation no optimiser runs: the front end interprets a fold one step at a time, and the optimiser that fuses Listing 5's stages into one residual pass sees only the code emitted for run time. So at compile time only a declared law shortens a fold, chosen by resolution, as for Listing 4's overloads: it buys at build time what the optimiser finds at run time by itself.

Not every algorithm converges, so Jlt allows a *budget*, a counter beside the structure's own count, as the compiler has one; exhausting the compiler's is not a value, and exhausting Jlt's is the empty optional, the third answer of Table 1 (Appendix G).[6]

A law can leave the compiler a batch. Listing 5's section names its law in its type, so one more overload of `count` takes no table: a window is independent pairs, and each position its own lane. Written as folds over small blocks, it vectorises at `-O2` with no intrinsic, about five times the table's speed (Appendix G); the overload names a law

[6] The compiler's budgets apply at compile time only, Jlt's at both binding times, and they are the compiler's to set: raised, the refused fold of 10^8 copies compiles and holds.

```
// Σ, the alphabet :
constexpr Alphabet bases = abc("ACGT");
// Σ* = ⋃ₙ Σⁿ :
constexpr Words oligos = words(bases);
// ~rev, the reverse complement: CCTA ⊑ TAGG
// { w ⊑ Σ⁶ | ~rev w = w }, the palindromic
↪  hexamers :
constexpr auto pali6 =
  oligos & len(6) & fixed_by(~rev);
// { (s, i) ⊑ Σ* × ℕ | s[i, i+6) ⊑ pali6 } :
constexpr auto cuts(size_t N) {
  return (oligos & len(N)) » pali6;
}

static_assert(
  count(cuts(12)) == 1'835'008uz &&
  count(cuts(32)) == 27uz « 58);
```

```
// those of pali6 beginning GA :
constexpr auto sites =
  pali6 & prefix("GA");

// the table, computed once by the compiler,
// a function of any s :
constexpr auto scanner =
  *automaton(sites);
static_assert(states(scanner) == 11uz);

// and only now a sequence :
constexpr auto dna = bytes(
  "GAATTCGAGCTCGGTACCCGGGGATCC"
  "TCTAGAGTCGACCTGCAGGCATGCAAGCTT");

// ten by definition, in windows of six :
static_assert(count(dna » pali6) == 10uz);

// two by the table, in one pass :
// EcoRI's and SacI's
static_assert(
  count(dna » scanner) == 2uz &&
  sites(bytes("GAATTC")) &&
  sites(bytes("GAGCTC")));
```

```
-- the right panel's query at every N-mer, the
-- bind of the bag monad, counted by its laws:
-- a cylinder per offset, 4^3 · 4^(N−6), N − 5 alike
cuts N = Σ^N »= (λ s. s » pali6)
count (cuts N)
      = (N − 5) · 4^3 · 4^(N − 6)
count (cuts 32) ~> 27 · 2^58
       -- a constant; no sequence built
```

```
-- the table is a closed term,
-- the pass over the sequence is not
automaton sites ~> (0, T, F)
  -- static, 11 states: the compiler's
count (s » scanner) =      -- the residual
  π₂ (fold (λ (q, c) a. (T q a, c + F (T q a)))
           (0, 0) (map rank s))
automaton pali6 ~> (0, T', F')
  -- 277 states: past the budget, so
  -- the same function runs at run time
```

Listing 5 Palindromic DNA restriction sites, counted two ways on one definition. A palindrome is a word equal to its reverse complement, as EcoRI's GAATTC; an enzyme cuts where one occurs. Left: how many cuts a cocktail of every six-cutter makes in all 4^N N-mers, each digested once. Right: the cuts of a smaller cocktail, the six-cutters beginning GA, in any one sequence, the caller's bytes read where they lie; here the polylinker of the plasmid pUC19. Below each: its translation (Section 4), a closed term the compiler reduces and a residual for the run time. Without the count's laws, the left panel would build the N-mers and exhaust the step limit (Table 5) before reaching its **static_assert**. The table is Brzozowski's automaton [14], for one site the matcher of Knuth, Morris and Pratt [56], which partial evaluation derives [22]. Its cost is moved to compile time where the budget admits: "negative overhead abstraction" [97].

of biochemistry and no instruction of any machine, and C++26's std::simd [58] would make the lanes explicit.

The same laws license parallelism. A parallel concat is one more overload, licensed by IsCommutativeMonoid. The branch on the binding time that selects it desugars to its compile-time branch, because the same law proves the branches agree (Appendix B).

C++ passes no dictionaries but instantiates, so the translation models what a constrained call means, not what the compiler emits. Instantiation is partial evaluation of the dictionary-passing term at a static type, so the two agree on every closed term, and parametricity holds of every instantiation (Appendix G).

7 Known Limitations

Table 1 says which instrument holds each rule at build time.

7.1 Limits of principle

The compiler does not check a template's body against its constraints. Definition checking was the half of the C++0x design [39] that C++20 did not take up (Section 1). So Rules 2 and 3 are posture: they are held outside the compiler, and no tool turns them into the compiler's verdict.

Lookup stops at std. A name in a template is looked up twice: at the template's definition, among the declarations visible there, and at instantiation only by argument-dependent lookup, in the namespaces of the argument types, which for a standard type is std, where no instance of ours lives. So an instance declared after the shape is found only as a specialisation of a primary template the shape names, as Listing 2's deleted unit is, and a constrained overload must precede the shape. This is a constraint on declaration order that the translation's instance resolution does not have, and FJlt resolves it by fiat: an instance is a specialisation of the trait, found by the trait's name.

A shape knows the laws someone declared. C++'s own words are that a type *satisfies* a concept, which the compiler decides, and *models* it, which it does not [47, res.on.requirements], and the Core Guidelines ask that a concept carry meaningful semantics [101, T.20].[7] A law is what modelling asks beyond satisfying, and the posture makes it a declared certificate, a specialisation the compiler finds but does not check against the operation. So false means *not known*, not *no*, and the same

[7] The standard calls the first kind of requirement syntactic and the second semantic. That split is between what the compiler decides and what it leaves to the programmer, and it does not coincide with the split of programming-language theory between static and dynamic semantics [80], whose static semantics, type checking, falls on the syntactic side here.

declaration is the only refusal available for a carrier that passes a shape by signature and breaks its law by value.[8]

Undefined behaviour has one binding time. Constant evaluation refuses it and run time does not (Table 5), the one way the two evaluations of Theorem 2 can disagree in C++ where they cannot in FJlt, and the reason the listings' carriers are unsigned. A Jlt program on **int** is a Jlt program only where its arithmetic cannot overflow, a fact no rule checks.[9]

The compiler has its own limits. Table 5 lists two, steps and call depth, and the template instantiation depth bounds both binding times. Rule 3 keeps them predictable, because every fold's length is in the program: a count, which meets no depth limit, or the depth of a term, one instantiation or one call per nesting.

7.2 Limits of this presentation

The linter is sound and incomplete. The repository's linter, run in continuous integration on the compiler's own front end and documented there, checks Rule 3 for every instantiated body, Rule 1's syntactic part, and Rule 2 at the level of names: a body may use, on a value of parameter type, only what a declared constraint names. A finding is a breach or a gap in its reading of the standard's concepts, and its silence is not a proof. Review complements it: an agentic reviewer reads every pull request against the three rules, and the listings and the library's core are held at zero on all three to the extent the two instruments reach.

The adequacy argument covers part of the library. Tables 6 to 8 are measured complete for the category module and the listings only. Other modules are held by the same linter at a count that only goes down.

The standard library is trusted. The listings compile with Clang 23 against two standard libraries, trusted as every C++ program trusts them and its compiler [108]; the youngest piece they use, the folds over ranges, is complete since libc++ 23 [107].

8 Related Work

Type classes. Dictionary passing is Wadler and Blott's [113]. Table 4 is that translation with C++'s names in the left column, and the row without a counterpart there, structural satisfaction, is a difference Bernardy et al. [8] weigh in their comparison of

[8] * on **size_t** is commutative, but nobody in the listings certifies it, so IsCommutativeMonoid<**size_t**, multiplies<**size_t**» is refused. **double** under + has the signature of a monoid, but + is not associative and the unit law fails at NaN, and nobody certifies it.

[9] A signed integer is a chain under < and a bounded lattice under min and max, but no monoid under + or *, since overflow is undefined.

concepts and type classes. Coherence, that a program has one meaning however its constraints are resolved, is where the designs diverge: Racordon et al. [84] survey how Haskell, Swift, Rust and Scala keep or trade it, between unique instances and context-dependent resolution. FJlt takes the strict side by construction, since an instance is a function specialisation and the one-definition rule admits one per head. Theorem 3 is that choice stated as a property of the translation, and the price, that instances for families of types declared after their shape do not fit, is in Section 7. Instance resolution is the inference half of type classes, Hindley–Milner's [26, 71] extended with predicates [48]; FJlt infers no types, since C++ deduces them, but it infers laws the same way. Rust and Swift also check a generic body against its bounds: a comparison on an unconstrained type parameter is a compile error in both, and Rust's message names the missing bound [5, 52]. That is Rule 2 as a compiler check, the half C++20 dropped, and the posture, together with the linter of Section 7, is what recovering it by hand costs.

Concepts, and the calculus next door. The nearest neighbour of FJlt is Siek and Lumsdaine's F^G [92, 93]: System F with concepts and models, translated into System F by dictionary passing, with the translation proved sound in Isabelle. It grew from the same work as the concepts then proposed for C++0x [30, 39], which checked template bodies and were removed [98], leaving the gap Rule 2 names. F^G's models are nominal and can be declared for whole families of types, which is what the library behind this paper's listings needs after its shapes, where FJlt's function-specialisation instances are not found (Section 7). FJlt is the smaller calculus, and the price of being a subset of a shipped language. Of the generic-programming features Garcia et al. [34] compare across eight languages, two that C++ then had no explicit support for, multi-type concepts and retroactive modelling, are what Listing 2 exercises.

Featherweight calculi. Featherweight Java [45] set the pattern: a core calculus small enough to prove things about, with a syntax that is a subset of the host language's. Featherweight Go [40] follows the pattern for structural subtyping and monomorphisation, and *Generic Go to Go* [31] proves a dictionary-passing translation correct by a bisimulation. That is the genre Section 5 is in, preservation and simulation of a translation, with soundness, termination and parametricity inherited from the target rather than proved from scratch.

Formal semantics of C++. CoreC++ [115] gives an operational semantics and a machine-checked type safety proof for multiple inheritance; Ramananandro, Dos Reis and Leroy [86, 87] mechanise object layout, construction and destruction in Coq. These formalise the part of C++ that Rule 1 excludes, objects with identity and lifetime, and none covers templates, concepts or constant evaluation, so FJlt is a fragment with its own grammar rather than a subset of a formalised C++.

What System F supplies. Strong normalisation is Girard's [37], with inductive types Matthes' [67], and parametricity Reynolds' [88]: the appendix uses both as black boxes, which is the point of translating into System F; data types as their own folds

is Böhm and Berarducci's reading [11], which the translation takes with iterators in place of Church encodings.

Functional programming in C++, and the other direction. Functional programming in C++ is practised and taught: in a book [24], in a metaprogramming library [29], and in category theory with examples in C++ [70]; none states a fragment or proves anything of it. The opposite route to the same end verifies a program in a proof assistant and extracts C from it, as Low* does from F* [82]; Jlt keeps the programmer in C++ and puts the proof in the translation.

Coding standards and profiles. The documents an audit of C++ works from are rule catalogues: MISRA C++ [72] and the SEI CERT standard [7] list hundreds of rules between them, each justified by the defects it prevents, with no statement of what a compliant program computes. Jlt is narrower, a pure computational core with no I/O, and so complements them rather than replacing them; where they meet they can differ, as MISRA forbids recursion except in constant evaluation and Rule 3 admits structural recursion on the strength of a termination argument. The nearest relatives are the safety profiles, now aimed at C++29 [103, 104], named sets of rules the compiler would enforce, and the criticism that a profile should come with the invariant it upholds [76]: Jlt reads as a functional profile, held by review and a linter rather than wholly by the compiler, and its three rules state theirs.

9 Conclusion

Three rules on standard C++23 were stated, and a program that keeps them is, up to the gaps Section 3 names, a term of a core calculus whose syntax is that C++. The calculus translates into System F by dictionary passing, and the translation preserves types and evaluation and is a function of the program (Theorems 1 to 3), so a program of the calculus terminates, is parametric and is sound (Corollaries 1 to 3).

Two of the three rules are held by the author, a linter and an agentic reviewer, not by the compiler (Section 7).

Four things follow as work not done here. Completing the linter into a checker, with definition checking over types, arities and the full constraint language, as the C++0x design had it. Certifying that a run-time shortcut agrees with the fold it replaces, which a branch on the binding time leaves to a law or a reader: the third homomorphism theorem characterises when a fold can be split [36], and a shape could declare it. A mechanisation of the proofs, the aspirational end, as Siek and Lumsdaine did for F^G in Isabelle [93]. And laws that are computed with rather than only declared, which needs terms and their composition.

Declared laws, and the laws inferred from them, license optimisations of the application's own algebra, as types rule out errors: by a sound check, not a decision of equivalence. The same laws are those a query optimiser applies to relational algebra [1, Chapter 6], so the posture reads, the other way round, as an intensional SQL: queries over relations given by their laws, not by their rows. Where that reading

should pay most is a hint, not a claim. In C++23 nothing allocated during constant evaluation outlives it, so the persistent structures of functional programming, built on the heap, do not cross to run time. C++26 lets an array through [20]. What crosses is state whose size is fixed at compile time, and the computations that specialise well across it may be those that run in small state over an unbounded input, the streaming algorithms [3]: Listing 5's matcher is one, a table that crosses and a pass that keeps a state and a count. Streaming queries, and the plans an optimiser chooses for them, are where an intensional SQL in this posture would be tried first. The laws survive the trip into Python, up to the compile-time evaluation: a session that composes and counts through the bindings runs the overloads the laws license, for those who would rather tinker than compile. Nothing was added to the language, the compiler or the library: what was added is a boundary, inside which a mainstream compiler checks claims of a modest but real kind about programs that then run unchanged, for the people who already write C++23.

Acknowledgements The author wishes to thank Bernhard Kräutler, Gisela Reichmuth, Simon Schwarz, and Ivanna Smolyana for interesting discussions and critical feedback.

Use of Agentic Systems and AI-Assistance Disclosure

This paper, its appendices and the code it reports on relied substantively on AI assistance, disclosed here as the journal's policy asks: 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; every mistake that remains is the author's.

Tools, and how each was gated. Anthropic's Claude, the Opus and Fable models through the Claude Code CLI, did most of the work, including cold reads of the draft in sessions without the writing session's context. It worked largely autonomously, in issues, pull requests, slices and commits, in an environment made deliberately constrained and antagonistic: strong type checking, substantial test coverage, formatters, Doxygen and the linter of Section 7; every pull request was gated by continuous integration and by review, usually one by the author and several by Copilot, and merged by hand. GitHub Copilot's reviewer was used for pull-request review only, in the deepest mode available, with the draft paper as context; its findings on the metatheory of the appendices led to the declaration order, the restriction to total primitives, the small-step evaluation and the present order of the proofs. Google's Gemini answered quick questions of the research kind, proofread, and read the appendices' proof sketches; its reviews needed heavy discounting, gave a few serendipitous finds, and were fed back to Claude with the author's opinion of them. The bibliography was used to attune the context and as a source of authority, and builds were offloaded to continuous integration to keep them reproducible, visible and on the record.

Modes. The assistants were used in agreement, to help structure a hypothesis, and in opposition, asked to refute or weaken one; the cold reads, the review threads and

the second opinions were of the second kind, and the author decided what survived them.

Tasks and places. Draft generation of prose, listings, tables and figures; the appendices' statements and proofs, drafted from the design decisions recorded in discussion and revised by the author; the test that holds the listings; verification of every citation against its source; and responses in review threads. Every section and appendix: no part was written without assistance, and none accepted without the author's reading. The repository [57] and its pull requests hold the record: commit messages, review threads and, while GitHub keeps them, build logs.

A Featherweight Jlt syntax

P	::=	$\overline{D}\ e$	program
D	::=	template <typename $\overline{\alpha}$> concept C = $\overline{A}$;	shape
	|	template <typename $\overline{\alpha}$> constexpr $\tau\ f(\overline{\tau})$ = delete;	trait
	|	template <> constexpr $\tau\ f$<$\overline{\tau}$>($\overline{\tau\ x}$) { return e; }	instance
	|	template <typename $\overline{\alpha}$> requires $\overline{Q}$ constexpr $\tau\ f(\overline{\tau\ x})$ { return e; }	generic function
	|	template <typename $\overline{\alpha}$> struct O { $\overline{\tau\ y}$; template <typename $\overline{\beta}$> requires $\overline{Q}$ constexpr τ operator()($\overline{\tau\ x}$) const { return e; } };	function object
A	::=	Q | e	nested shape, law
	|	requires ($\overline{\text{const } \tau\ x}$) { $\overline{\{\ e\ \}\ \text{->}\ S\text{<}\tau\text{>};}$ }	required calls
Q	::=	C<$\overline{\tau}$> | !C<$\overline{\tau}$>	constraint
τ	::=	α | b | T<$\overline{\tau}$>	type
e	::=	x | c | f | f<$\overline{\tau}$>($\overline{e}$) | $f(\overline{e})$ | O<$\overline{\tau}$>{$\overline{e}$} | $e(\overline{e})$	term
	|	$\ominus e$ | *e | $e.y$ | $\tau(e)$ | $e \oplus e$ | e ? e : e	built-ins, by arity

Figure 3 The abstract grammar [80] of FJlt, a subset of C++23, namespace qualifiers elided. Terminals are in monospace, and an overbar abbreviates a sequence. A letter with a row has the structure the row gives; a letter with no row is a name or a symbol, listed below. P is a program. D is a declaration. A is a requirement. Q is a constraint. τ is a type. e is a term.

FJlt, the kernel of Figure 1 and what Section 3 desugars to, is the subset of C++23 that Figure 3 writes, in the style of Featherweight Java [45]. Its base types and their **constexpr** operators form a many-sorted algebra [10, 90] over a signature Σ. Loosely speaking, a type such as **bool** or **size_t** is a *carrier set* if it keeps Rule 1 (Substitution), and an operator on such types, $\ominus$ or $\oplus$ in the figure, is a *function* between them if it is

total. It need not stay on one carrier set, as == on std::string gives a **bool**. Listing 6 is a valid example snippet, read in the figure's terms.

```
// D: a trait; its primary stays deleted
template <typename T>
constexpr T op(T, T) = delete;

// D: an instance; + an operator of Sigma
template <>
constexpr std::string
op<std::string>(std::string a, std::string b) {
 return a + b;
}
```

```
// D: a shape; A its required calls
template <typename T>
concept IsMagma =
   requires(const T a, const T b) {
    { op(a, b) } -> std::same_as<T>;
   };

// D: a generic function; Q its constraint
template <typename T> requires IsMagma<T>
constexpr T twice(T x) {
 return op(x, x);
}

// e: the closed term the program evaluates
twice("ab"s) + "c"s == "ababc"s
```

Listing 6 A small program, each declaration labelled with its form in Figure 3; its ::s literals are sugar.

In Figure 3, α, β range over type variables, b over base types, T over type constructors, declared or adopted (Table 9), O over the structs among them, function objects, standard (std::plus, …) or declared, std::pair and std::optional, y over their fields, $\ominus$ and $\oplus$ over the unary and binary operators of Σ, C over concept names, the standard ones included, S over the result-type concepts std::same_as and std::convertible_to, f over function names, x over term variables and c over constants. A template is always fully applied: α ranges over types, and there is no template template parameter, so System F suffices.

B From Jlt to Featherweight Jlt

This appendix is the —sugar arrow of Figure 1, which Section 3 only outlines: each construction of C++ that a Jlt program writes beyond the grammar of Appendix A, with its desugaring into the kernel and its condition, and no proof. A condition is on how a term is built, never on inspecting it afterwards, and the rules are asked of the kernel term. Three reasons cover the constructions: one is only a spelling, one keeps its state private, or one is a choice. In that order the condition passes, on the whole, from the compiler to the repository's linter to a reader. The listings are FJlt up to this sugar, reading **static_assert**(e) as the claim that e evaluates to true, with two exceptions: Listing 3's Collatz chain is finite by fact, not by a condition on $\mathscr{R}$, and Listing 5 dereferences an optional in a constant initialiser, admitted under *A choice* below. An FJlt program is a Jlt program with the same meaning, provided no law and no **static_assert** builds a box (Appendix E).

construction	FJlt, provided	image	the condition held by
`auto`, `decltype`, argument deduction	nothing	the type the compiler deduces	the compiler
alias, `tuple`, `array`	the parts FJlt	substitution, nested pairs, a range of $\mathscr{R}$	the compiler
variable or class template	each member of the family an FJlt instance	instance with a context, a default beside families resolved by specificity	the compiler, by lookup
const member function	`const`, FJlt body	generic function with the object as first parameter	the linter
constructor, conversion operator, default argument, designated initialiser	`constexpr`, FJlt body	a function returning the struct; a declared conversion; an overload with the argument supplied; the fields in order	the linter for `constexpr`, a reader
explicit conversion	a total conversion between carriers	a declared function	a reader
an operator on a constrained type	a constraint names the standard function object the standard defines by it, `plus<T>` for `+`, and the operator has type `T` there (`true + true` is the `int` `2`)	that object's call	the linter for the name, a reader for the type
operator overloading, user-defined literal	FJlt body	a named function call	the compiler
non-type template parameter	the value a constant of a carrier	a type per value, as `integral_constant` spells it	the compiler
overloading	distinct carriers, or constraints ordered by specificity	specialisations of one trait, or generic functions	the compiler, by ambiguity
`friend`, access specifier	a friend an FJlt function	a free function; nothing	the compiler
inheritance	every base an empty class, no virtual function	a registration, read as the fact it states	the linter
generic `cata` over a term type	algebra and term FJlt	the kernel's primitive `cata`	the compiler

Table 6 Constructions that are another spelling of a kernel term. In each row, a term built by the construction from FJlt terms, under the condition, is the FJlt term in the image column.

construction	FJlt, provided	image	the condition held by
lambda	captures by copy of FJlt values, FJlt body	function object, captures as fields	the linter for the captures, a reader for the image
local constant, structured binding	the initialiser FJlt, the name bound once	a β-redex; a binding as .first, .second	the linter
local state that does not escape	assignments only to locals, by-value parameters, elements of a local array, or a builtin's out-parameter that is a local; nothing written observable after the call	the state threaded as values, one binding per assignment, as runST encapsulates it	the linter for escape, a reader for the rest

Table 7 Constructions whose state stays inside a call, read as Table 6.

Only a spelling. Most of what a program writes beyond the grammar is a kernel term spelt another way, and most of Table 6 needs no further word. A variable template v<T...> = e is the nullary function template v<T...>() { **return** e; }, and a class template with a static constant member is the same, since constant evaluation of the three agrees. A partial specialisation over a family is an instance with a context, in the grammar. A primary that answers false beside families that answer true is two generic functions whose heads both match, which resolution by specificity orders. Where lookup (Section 7) leaves the FJlt spelling unwritable, the host's spelling is FJlt by this statement. A **const** member function τ m(τ x...) **const** { **return** e; } of a function object is the generic function m(O self, τ x...) whose body reads the fields of self where the member read **this**; Rule 1 supplies the **const**. A static member function is a generic function. A struct with fields and no call operator is a product, nested pairs. std::array<T, n> is a product of n copies and also a member of $\mathscr{R}$, with length n, a head and a tail, so a fold over it is already in the grammar. Constrained overloads are generic functions, and where their heads both match, resolution by specificity orders them, as for Listing 4's overloads of fold and concat. Jlt has no subtyping: the grammar has no base clause and no derived-to-base conversion, a class with a base that holds state or a virtual function is outside the posture, and the linter's inheritance check holds the module at zero. A base clause on an empty class, such as std::true_type, is a registration: the family row spelled by inheritance, read as the fact it states.

Private state. A capture or a local stays inside the call, so nothing outside can tell it from a value passed along. A lambda [c...](**const** τ x...) { **return** e; } whose captures are FJlt values taken by copy and whose body is an FJlt term under those captures and parameters is the function object with the captures as fields and e as the body of its call operator. A generic lambda written with a template head and a **requires** clause is the function object with a generic call operator. That is the standard's own definition of the closure type: a class with the copied captures as members and a **const** call operator, the function object production of Figure 3 to the letter. Rule 1 excludes capture by reference. A capturing closure type is copy constructible and

construction	FJlt, provided	image	the condition held by
if and else on a value	both branches return FJlt terms of one type	`p ? A : B`	a reader
if constexpr on a value	condition a closed bool term, both branches FJlt of one type under the same constraints	`p ? A : B`	a reader, by the rewrite
enumeration, switch	every enumerator handled, each case an FJlt term of one type	a finite sum of units and its nested case	the compiler, by its switch warning
sum	branches FJlt on the summands	the case form	the kernel, a reader for a guard
if constexpr on a constraint	then-branch FJlt under the constraint, else-branch under its negation	an instance chain, the more constrained generic function preferred	the compiler at concrete types, the linter at a type variable
a call its clause does not decide	the caller instantiates at a concrete type	constraint inference: the call's resolution a parameter the caller supplies	the linter reports it
if consteval, or an if on is_constant_evaluated()	both branches compute the same value on every input	the compile-time branch	the declared law that licenses the other branch, or a reader

Table 8 Constructions that choose, read as Table 6.

neither default constructible nor assignable, so it is not std::semiregular and Rule 1 does not admit it as a value; it is admitted as sugar, judged through its image, which is semiregular when its captures are. A closure type has no name, so a lambda is never a carrier and never the head of an instance, and nothing in the statement asks to look inside it once built.

A choice. A branch is what a spelling cannot hide, and it needs one of the kernel's own forms: ?:, the case form, or generic functions ordered by specificity. A body **if constexpr** (p) **return** A; **else return** B; with p a closed **bool** term, a law or a trait read, and A, B FJlt terms of one type under the same context, is the term `p ? A : B`. The condition is stronger than C++ asks, since C++ does not instantiate the discarded branch and FJlt types both, and the compiler decides it: the rewritten body compiles or it does not. Constant evaluation of the two spellings agrees because both evaluate p first and then one branch. Whether the two branches share a type is decided per site by the rewrite. A site whose branches differ in type is a branch on types in a value's clothing.

The kernel has the case form (Table 9), so given FJlt terms f and g on the two summands their copairing is an FJlt term, case(s, f, g). has_value and value_or on an optional are that case on $1 + T$, and std::visit(O{}, v) is the case whose branches are one generic call operator entailed at each summand. * on an optional is partial, undefined on the empty one. It is admitted under a has_value guard, which desugars to the case form, as the head of a range is admitted only when its length is positive. It is also

admitted in a constant initialiser, where constant evaluation refuses the empty case and the compiler holds the guard, as for Listing 5's *automaton(sites). std::get on a variant is partial too and is not admitted: a variant is read by the case form.

A choice on a constraint. A body **if constexpr** (C<T>) **return** A; **else return** B; in a generic function with head $\overline{Q}$, where A is an FJlt term under $\overline{Q}, C\texttt{<T>}$ and B one under $\overline{Q}, !C\texttt{<T>}$, is the pair of generic functions with those two heads. The pair plays the role of a two-clause instance chain of Morris and Jones [73], an explicit alternation decided by one constraint, though resolution by specificity (Appendix C) rather than clause order picks its branch. At a concrete type the compiler decides $C\texttt{<T>}$, and at a type variable the call resolves only where the enclosing context decides it. When the test is std::same_as<T, X> at a concrete X and T is the whole head, the first function is an explicit specialisation at X. A test of a **requires** expression written in the body is not a declared constraint under Rule 2, and the linter's inline-requires check holds the module at zero. This split and the next desugaring add declarations, so they are not local in Felleisen's sense [32].

A call its clause does not decide. A generic function whose body calls a chain at its own type variable, without a clause that entails the chain's constraint or its negation, varies with the type through a constraint it did not declare. C++ accepts it and resolves the call at each instantiation; the kernel refuses it, by GHC's rule. Its desugaring is constraint inference: the call's resolution becomes a parameter of the enclosing function, which each caller supplies after resolving it at a concrete type, where resolution is a function, so the kernel term agrees with what C++ instantiates. The price is in the free theorem, which now ranges over that parameter. Written by hand, the same thing decides the constraint in the clause, by requiring it or by splitting on it.

A choice on the binding time. **if consteval**, and an **if** on std::is_constant_evaluated(), are how a program can observe its binding time, and FJlt need not have them. **if consteval** { A } **else** { B } desugars to A, provided B computes the same value as A on every input, so that the program cannot observe the difference after all (Table 8).[10] A parallel concat is the case in point: it is one more overload, a run-time one, since the policies' overloads are not **constexpr**. std::reduce may regroup and reorder its operands, so its result is determined only for an operation that IsCommutativeMonoid certifies, and Rule 1 gives the absence of data races its parallel policies ask. For the parallel concat the condition is the free monoid's universal property: a fold over a concatenation is the operation applied to the folds of the parts, by associativity, and commutativity lets the parts come in any order. So the law that licenses the shortcut also proves that

[10] The choice of A is not arbitrary: it is the branch constant evaluation takes, which is FJlt's evaluation, while B is typically not constant-evaluable, which is why the program branches at all.

the branches agree. For an arbitrary branch the condition is semantic, and a reader holds it, as for the part of Rule 2 the linter does not reach.

Why that is all of it. The tables are complete by measurement, not by argument, at the resolution of a syntax node's kind. The repository's linter lists every kind the category module and the listings write, leaving out the compiler's implicit wrappers, and each is the kernel's or a row, with if consteval added for the parallel concat. A kind does not tell every use apart: a binary operator is arithmetic or an assignment, and the assignments are the row of local state, which Rule 1's footnote admits and which reading, not counting, finds. With the tables closed, the second direction of Section 3 reads: a Jlt program built from the grammar by these constructions under their conditions is an FJlt program.

C Featherweight Jlt typing

Figures 4 and 5 say which FJlt programs are well formed, the domain of $[\![\cdot]\!]$ in Figure 1 and Section 4. As Appendix A's carrier sets keep Rule 1 (Substitution), their rules hold a program to Rule 2 (Abstraction), through the definition check and resolution, and to Rule 3 (Descent), through well-founded declarations and ranges (Section 2). There are three judgments. $\Delta \Vdash Q$ says that the constraint Q is *entailed* in the context Δ, which holds type variables and assumed constraints. $\Delta; \Gamma \vdash e : \tau$ types a term. $\vdash D$ says a declaration is well formed, and a program is well formed when its declarations are and its closed term has a type. The compiler holds a program to most of this. It does not hold a generic body to its constraints (Section 7), a call at a type variable to the resolution below, a body to the declared type of a converted result, or a range to its conditions, and those rest on the programmer's discipline.

Entailment and laws. E-Law is where a law is a value: entailment evaluates it, so a shape with a law is satisfied only at instances whose certificate holds. Laws are closed terms of base type, which keeps typing and evaluation from chasing each other. A law is evaluated where a record is built, at concrete types; at a type variable it is entailed only through an assumed shape that lists it, by E-Law-Assumed, as a call is through T-Req-Assumed. A negated constraint !C<T> carries no evidence: E-Not holds at concrete types where C does not, and the well-founded order below makes its negative premise decidable. The side condition of E-Shape makes entailment syntax-directed: an assumed shape is its record variable and is never rebuilt field by field, which is what Theorem 3 needs.

The definition check. D-Generic types a generic body under its constraints, and T-Req-Assumed is what makes that a check. D-Op is the same with a function object's fields among the variables, and D-Instance the same with an empty context. Inside a constrained body, a call on a constrained type is typed only as an instance of a requirement of a constraint in Δ. Op{}(x, x) is the pattern { Op{}(a, b) } at $a, b := x$, and unit<T>(Op{}) is the pattern { unit<T>(Op{}) }, whose parameters are the variables of

$$\text{(E-Assume)}\quad \frac{Q \in \Delta}{\Delta \Vdash Q}$$

$$\text{(E-Law)}\quad \frac{\varnothing;\varnothing \vdash e : \mathsf{bool} \qquad e \Downarrow \mathsf{true}}{\Delta \Vdash e}$$

$$\text{(E-Law-Assumed)}\quad \frac{\mathsf{concept}\ C = \overline{A} \qquad C\langle\overline{\tau}\rangle \in \Delta \qquad e = A_i[\overline{\tau}/\overline{\alpha}]\ \text{a law}}{\Delta \Vdash e}$$

$$\text{(E-Shape)}\quad \frac{\mathsf{concept}\ C = \overline{A} \qquad C\langle\overline{\tau}\rangle \notin \Delta \qquad \Delta \Vdash A_i[\overline{\tau}/\overline{\alpha}]\ \text{for each}\ i}{\Delta \Vdash C\langle\overline{\tau}\rangle}$$

$$\text{(E-Op)}\quad \frac{\Delta;\overline{x : \tau_x} \vdash e : \tau' \qquad \tau'\ S\ \tau}{\Delta \Vdash \mathsf{requires}\ (\overline{\mathsf{const}\ \tau_x\ x})\ \{\ \{\ e\ \}\ \text{->}\ S\langle\tau\rangle;\ \}}$$

$$\text{(E-Not)}\quad \frac{\overline{\tau}\ \text{concrete} \qquad !C\langle\overline{\tau}\rangle \notin \Delta \qquad \Delta \not\Vdash C\langle\overline{\tau}\rangle}{\Delta \Vdash !C\langle\overline{\tau}\rangle}$$

Figure 4 The rules of entailment, $\Delta \Vdash Q$. Six rules: an assumption, two for laws, a shape, a requirement and a negation.

the **requires** clause free in it, here none; a trait is reachable the same way. A variable of a fieldless function-object type denotes that type's one value, so `op` of type `Op` is `Op{}`, which is how Listing 3's `concat` applies its `op`. The requirements of a shape, in T-Req-Assumed and E-Law-Assumed, are its own and, transitively, those of its nested shapes: an assumed `IsCommutativeMonoid<T, Op>` yields the operations and the associativity law of `IsMonoid<T, Op>`. Where more than one assumed requirement matches, the rules take the leftmost shape of the **requires** clause and, within it, the first matching requirement in declaration order, so both rules are functions of the body; once instantiated the candidates denote one operation, by Lemma 3. A premise $f : \sigma \times \tau \to \sigma$ in T-Fold and T-Case says that f is a function name or a function object whose application to arguments of those types is typed at σ by T-Call, T-Trait, T-Op, T-App or, at a type variable, T-Req-Assumed.

Resolution. A trait call $f\langle\overline{\tau}\rangle$ at concrete types *resolves* to a declaration of f in two steps, as C++ resolves an overloaded template. The first step takes the most specific of the generic functions whose head matches $\overline{\tau}$ and whose context is entailed, the deleted primary among them. If that is the primary, the second takes its explicit specialisation at $\overline{\tau}$ if there is one. The deleted primary never resolves, which is how an operation nobody registered fails a shape. Well-formedness requires resolution to be a function, in two cases. At concrete types, no two specialisations share a head, and among the generic functions whose head matches and whose context is entailed there is a unique most specific one, by head and then by context, as C++ orders constrained templates; C++ rejects the program where there is none. At a type variable, in a body with context Δ, a call resolves only when its choice survives every instantiation of the body: every candidate, explicit specialisations included, whose head unifies

Calls

$$\text{(T-Op)}\quad \frac{O\langle\overline{\tau}\rangle : \overline{\sigma} \to \sigma \text{ (the standard's signature)} \qquad \Delta;\Gamma \vdash \overline{e : \sigma}}{\Delta;\Gamma \vdash O\langle\overline{\tau}\rangle\{\}(\overline{e}) : \sigma}$$

$$\text{(T-Conv)}\quad \frac{\Delta;\Gamma \vdash e : \tau' \qquad \tau' \text{ converts to } \tau}{\Delta;\Gamma \vdash \tau(e) : \tau}$$

$$\text{(T-App)}\quad \frac{\Delta;\Gamma \vdash e : O\langle\overline{\tau}\rangle \qquad O\text{'s call operator is } \forall\overline{\beta}.\overline{Q} \Rightarrow \overline{\tau_x} \to \tau_r \qquad \Delta \Vdash \overline{Q}[\overline{\tau}/\overline{\alpha}][\overline{\sigma}/\overline{\beta}] \qquad \overline{\Delta;\Gamma \vdash e' : \tau_x[\overline{\tau}/\overline{\alpha}][\overline{\sigma}/\overline{\beta}]}}{\Delta;\Gamma \vdash e(\overline{e'}) : \tau_r[\overline{\tau}/\overline{\alpha}][\overline{\sigma}/\overline{\beta}]}$$

$$\text{(T-Call)}\quad \frac{f\langle\overline{\sigma}\rangle \text{ resolves to } f_i : \forall\overline{\alpha}.\overline{Q} \Rightarrow \overline{\tau_x} \to \tau \qquad \Delta \Vdash \overline{Q}[\overline{\sigma}/\overline{\alpha}] \qquad \overline{\Delta;\Gamma \vdash e : \tau_x[\overline{\sigma}/\overline{\alpha}]}}{\Delta;\Gamma \vdash f\langle\overline{\sigma}\rangle(\overline{e}) : \tau[\overline{\sigma}/\overline{\alpha}]}$$

$$\text{(T-Trait)}\quad \frac{f\langle\overline{\sigma}\rangle \text{ resolves to a declaration of type } \overline{\tau_x} \to \tau \qquad \Delta;\Gamma \vdash \overline{e : \tau_x}}{\Delta;\Gamma \vdash f\langle\overline{\sigma}\rangle(\overline{e}) : \tau}$$

The definition check

$$\text{(T-Req-Assumed)}\quad \frac{C\langle\overline{\tau}\rangle \in \Delta \qquad \{\ e_r\ \}\ \texttt{->}\ S\langle\tau\rangle \text{ is a requirement of } C \text{ with parameters } \overline{a : \tau_a} \qquad \Delta;\Gamma \vdash \overline{e : \tau_a}}{\Delta;\Gamma \vdash \tau^S(e_r[\overline{e}/\overline{a}]) : \tau}$$

$$\text{(D-Generic)}\quad \frac{\overline{\alpha},\overline{Q};\ \overline{x : \tau_x} \vdash e : \tau}{\vdash \texttt{template <typename } \overline{\alpha}\texttt{> requires } \overline{Q}\ \texttt{constexpr}\ \tau\ f(\overline{\tau_x\ x})\ \{\ \texttt{return}\ e;\ \}}$$

Recursion, from the adopted library

$$\text{(T-Fold)}\quad \frac{\Delta;\Gamma \vdash r : R\langle\tau\rangle, R \in \mathscr{R} \qquad \Delta;\Gamma \vdash a : \sigma \qquad \Delta;\Gamma \vdash f : \sigma \times \tau \to \sigma}{\Delta;\Gamma \vdash \mathsf{fold_left}(r, a, f) : \sigma}$$

$$\text{(T-Fold-R)}\quad \frac{\mathit{IsRange}\langle R, \tau\rangle \in \Delta \qquad \Delta;\Gamma \vdash r : R \qquad \Delta;\Gamma \vdash a : \sigma \qquad \Delta;\Gamma \vdash f : \sigma \times \tau \to \sigma}{\Delta;\Gamma \vdash \mathsf{fold_left}(r, a, f) : \sigma}$$

$$\text{(T-Cata)}\quad \frac{\Delta;\Gamma \vdash t : \mu F \qquad \Delta;\Gamma \vdash alg : \mu F \times \sigma^k \to \sigma \text{ at each node of arity } k}{\Delta;\Gamma \vdash \mathsf{cata}(alg, t) : \sigma}$$

$$\text{(T-Node)}\quad \frac{\Delta;\Gamma \vdash t : O\langle\overline{\tau}\rangle \qquad O \text{ a node of } F}{\Delta;\Gamma \vdash t : \mu F}$$

Figure 5 The rules for terms, $\Delta;\Gamma \vdash e : \tau$, and for generic functions, $\vdash D$. They come in three groups: calls, the definition check and recursion. Omitted as textbook: T-Var, T-Construct, T-Field, T-Inj, T-Case, and the rules for constants, pairs, boxes, conditionals and the arithmetic of base types.

with the call's types and that would be more specific than the chosen one is excluded by a constraint Δ decides, entailing it or its negation. That refines GHC's rule for overlapping instances [35], which looks at instance heads only and refuses to commit to an instance the types do not yet decide. C++ does not hold it: a call it would refuse is a body that varies with a type through a constraint it did not declare, a breach of Rule 2, which the repository's linter reports where it can see one (Section 7).

Well-founded declarations. Well-formedness also requires that the declarations, taken at their type arguments, be well founded under *depends on*. A shape at some types depends on the traits its requirements call and on its nested shapes, at those types. A trait at some types depends on the context of every candidate whose head matches there, which resolution asks even of a candidate it then rejects, and on the candidate it picks. An instance, a generic function or an operation at some types depends on the shapes of its context, on the declarations its body calls and on the operations its body constructs, and an operation also on the operations in its field, parameter and result types, each at the types it names. The relation is on instantiations and not on bare declarations, since a structured instance may ask for the shape that asks for it, at a smaller type: `IsMonoid` at a pair calls `unit` there, whose instance asks `IsMonoid` at the components (Section 4.1). The measure that falls is the pair of depth of the type arguments and declaration position. Resolution, entailment, E-Not's negative premise and E-Law's evaluation all recurse along that relation, so its being well founded makes the rules define one relation and makes resolution terminate. C++ has a rule of its own for a satisfaction that depends on itself, which the condition makes unnecessary. Two syntactic conditions suffice, and the repository's linter reads the second. A context on a structured head mentions only proper parts of its type arguments, so that each step descends by the head; compare the Paterson conditions [102]. The context of an instance whose head has no proper part at a slot, a type variable or a concrete type there, mentions only shapes declared before the trait's primary, so that a shape calling the trait is never asked for while the trait is resolved. A trait's specialisations may follow the shapes that call it, as Listing 2's do, since a specialisation depends on the shape only through its body or its context. A term type is required strictly positive: in the translated field types of its nodes the type occurs only directly, never under the domain of a function type. With one exception, a body calls only declarations it depends on.

The one exception. A function may call itself at the same instantiation on a value that visibly descends, Rule 3's step down. The argument in the slot of a parameter p is p/c with $c \geq 2$, $p \gg c$ with $c \geq 1$, or $p - 1$, under a test of p against 0 whose branch returns without the call. The parameter is of an unsigned carrier, or of a type variable under a shape the kernel reads as `IsCountWord`, whose fields are the operations the body uses and a coercion `{ size_t(n) } -> same_as<size_t>` that preserves the count, so a count no wider than `size_t`. This admits the binary method of Listing 4, whose `power` halves its count `n` of type `N`. The library's `cata` over typed terms is not this exception, as its recursion is on the type of the term and not on a value: it is the kernel's primitive `cata` spelled as a generic function, which is sugar (Appendix B).

Conversions. T-Conv types the conversion between carriers that C++ inserts implicitly and the kernel term writes, $\tau(e)$, total by the condition on conversions of Table 6. E-Op reads S as equality for std::same_as and as conversion for std::convertible_to. Write $\tau^S(e)$ for e when S is std::same_as and for $\tau(e)$ when it is std::convertible_to, so that T-Req-Assumed types τ^S of the pattern and a use of the requirement keeps its declared type when S-Call substitutes the instance (S-Conv). A body may therefore use a converted result only at the declared τ. A body that used the unconverted result, say to compare two of them at their own type, would be using an operation the constraint did not name, which the typing refuses and C++ accepts, and which the repository's linter reports where it can see it. The library needs the conversion form where a requirement's expression is an lvalue of the wanted type (**const** T& is not same_as<T>; IsPointed's unit), where a step returns a proxy for the carrier (HasNNOStep's successor), or where an object is presented by another type (Presents).

Ranges and terms. A range $R\langle\tau\rangle$ is an instance of a family $\mathscr{R}$ defined by conditions, a length $|r| \in \mathbb{N}$, fixed before a fold starts, and, when $|r| > 0$, a head and a tail $next\, r$ with $|next\, r| = |r| - 1$. T-Range types a range of $\mathscr{R}$ by its instance's conditions, iota(i, j) only at $i \leq j$, and S-Fold preserves them, since *next* does. T-Range-Op types size(r) at **size_t**, head(r) at τ and next(r) at $R\langle\tau\rangle$. The conditions are semantic, that a step descends or that two bounds are ordered, so a chain under a step that does not descend is ill-typed, and termination of a fold is in that measure assumed in the typing of its range. A range may be a type variable R under *IsRange*<R, T> = **requires** (**const** R r) { { size(r) } -> same_as<**size_t**>; { head(r) } -> same_as<T>; { next(r) } -> same_as<R>; }. The instances of $\mathscr{R}$ over τ, and only they, satisfy it at $T = \tau$: membership in $\mathscr{R}$, with its falling length, is a premise, so three operations without that law do not suffice. Listing 3's concat writes the weaker std::ranges::input_range<R>, which the kernel reads as *IsRange* with T the deduced ranges::range_value_t<R>; a standard range outside $\mathscr{R}$, an unsized view, is not in the kernel. T-Fold-R types a fold at a type variable R through the assumed shape *IsRange*<R, τ>: the length, head and tail are its three requirements, read from the record, and a caller's concrete range supplies the record by E-Shape at instantiation. In T-Cata, μF is a term type, the inductive type of the nodes that build it: a term is a finite composition of nodes, a node a function object whose fields are terms. The algebra is applied to each node together with the folded values of its legs, the node itself still in hand, which is the library's interface (alg(t, cata(alg, leg)...)) and the paramorphism's.

Inferred laws. A law at a construction is entailed, not declared: the law is a trait call, resolution picks the instance whose context the program entails, and E-Law evaluates its certificate, which must be true. Composed, these give a derived rule, the instance rule of the entailment of qualified types [48]:

$$\text{(E-Law-Ctx)}\quad \frac{\begin{array}{c} law\langle\overline{\tau}\rangle \text{ resolves to } \forall\overline{\alpha}.\,\overline{Q} \Rightarrow law\langle\overline{\tau'}\rangle \qquad \overline{\tau} = \overline{\tau'}[\overline{\sigma}/\overline{\alpha}] \\ \Delta \Vdash \overline{Q}[\overline{\sigma}/\overline{\alpha}] \qquad law\langle\overline{\tau}\rangle \Downarrow \mathsf{true} \end{array}}{\Delta \Vdash law\langle\overline{\tau}\rangle}$$

A finite power of a semiring is a semimodule over it in this way: the library declares the power a product of its base, and one instance, whose context is the base's semiring laws, gives the power's semimodule law, as semimodules over a semiring form a variety, closed under products. Each step reads the laws of a term's immediate parts only, so the inference is local [81], and it ends by the conditions on contexts that end instance search [102].

D Translation into System F

This appendix defines in full the arrow $[\![\cdot]\!]$ of Figure 1 and Section 4 on the programs of Appendix C, by dictionary passing [113]: a constraint becomes a record type, an entailment derivation a record, and a generic function a function of those records. The target is System F with products, sums, the sorts of Σ as constant types with its operators as constants, and the iterators of inductive types: $\mathit{iter} : \forall\sigma.\,\mathbb{N} \to (\sigma \to \sigma) \to \sigma \to \sigma$ and, for each term type μF, $\mathit{fold}_F : \forall\sigma.\,(F(\sigma) \to \sigma) \to \mu F \to \sigma$, each reducing one constructor at a time. Its iterative fragment embeds into System F by an encoding that preserves reduction [67]. Records are nested pairs.

Constraints and dictionaries. The translation's one non-local ingredient is the *dictionary* of an entailment: $[\![\Delta \Vdash Q]\!]$ is the record that witnesses Q, built by the same recursion as resolution. A dictionary projects along the path of nested fields, $q_C.m.r$ for a requirement r of the nested shape held in field m. A converted result is used only at its declared type (Appendix C), so the conversion the dictionary makes once, in the field, is the conversion the source would make at every use.

Declarations and terms. A function object is a pair of its fields and its call function, so a fieldless std::plus<**size_t**> is $((), \mathit{plus})$, passing it as a step uses its second component, and reading a field uses its first. A trait has no image of its own; its specialisations are closed terms $f_{\overline{\tau}}$, and a generic function that serves as an instance with a context is the function from records to a record that E-Shape builds its dictionary with. Where **size_t** is read as $\mathbb{N}$, by a coercion constant from the carrier to the inductive naturals, $[\![n]\!]$ is the natural number that is n's value; the translation of a term with a runtime n is the term with n as a free variable of type $\mathbb{N}$, and Theorem 2 is stated for closed terms. The call function of a constrained operation abstracts over its dictionaries as a generic function does, and an application supplies them. With one exception, below, a body calls only declarations it depends on, so the defined names are acyclic, and a term type is strictly positive, which the iterator and the normalisation results need.

The one exception. A descending self-call, the one exception of Appendix C, translates to a fold: along any path of calls each takes p to a smaller value and none is made at 0, so a path has at most $p + 1$ calls, the last for the base case. So $f = \lambda\overline{x}.\,(\mathit{iter}\ (\mathit{toNat}\ p + 1)\ F\ d)\ \overline{x}$ over all its parameters $\overline{x}$, p among them, the count read into the target's naturals by the carrier's coercion or the shape's field, where the successor does not wrap. $F = \lambda f.\,\lambda\overline{x}.\,[\![e]\!]$ is the body abstracted over the function its self-call names, a

types	
$[\![\alpha]\!] = \alpha \qquad [\![b]\!] = b$	
$[\![O\text{<}\overline{\tau}\text{>}]\!] = \overline{[\![\tau_y]\!]} \times (\forall\overline{\beta}.\, \overline{[\![Q]\!]} \to \overline{[\![\tau_x]\!]} \to [\![\tau]\!])$	a function object: its fields and its call function
$[\![R\text{<}\tau\text{>}]\!] = [\![R]\!] \qquad [\![\mu F]\!] = \mu F$ with $F\,X = \sum_O \prod \overline{[\![\tau_y]\!]}[X/\text{the leg types}]$	a range's state; a term type, the sum of its nodes' fields, a leg's type replaced by X
constraints, as record types	
$[\![C\text{<}\overline{\tau}\text{>}]\!] = \{\, \overline{r_i : [\![A_i]\!][\overline{\tau}/\overline{\alpha}]} \,\}$ where concept C = $\overline{A}$	one field per requirement
$[\![Q]\!]$ for a nested shape Q	its record type
$[\![!C\text{<}\overline{\tau}\text{>}]\!] = 1$	a negation carries no evidence
$[\![e]\!] = \mathit{Bool}$ for a law e	
$[\![\{\ e\ \}\ \text{->}\ S\text{<}\tau\text{>}]\!] = \overline{[\![\tau_x]\!]} \to [\![\tau]\!]$	an operation on parameters $\overline{x : \tau_x}$
dictionaries, from entailment derivations	
$[\![\Delta \Vdash Q]\!] = q$ if $Q \in \Delta$ with record variable q	E-Assume
$[\![\Delta \Vdash C\text{<}\overline{\tau}\text{>}]\!] = \{\, \overline{r_i = [\![\Delta \Vdash A_i[\overline{\tau}/\overline{\alpha}]]\!]} \,\}$	E-Shape
$[\![\Delta \Vdash e]\!] = \mathit{true}$	E-Law, E-Law-Assumed
$[\![\Delta \Vdash !C\text{<}\overline{\tau}\text{>}]\!] = ()$	E-Not
$[\![\Delta \Vdash \{\ e_r\ \}\ \text{->}\ S\text{<}\tau\text{>}]\!] = \lambda\overline{a}.\, [\![\tau^S(e_r)]\!]$	E-Op, over the requirement's parameters, the conversion in the field

Figure 6 The translation of types, constraints and entailment derivations; pairs, sums and boxes are the target's own (Table 9). A constraint becomes a record type and a derivation a record.

self-call at the same instantiation translating to $f\ \overline{[\![e]\!]}$ on that binder, with no type arguments and no dictionaries, which are those of the enclosing abstraction. d is the constant function to *default* of the result type (Figure 9), which at a type variable is the one field of std::semiregular's record that the translation uses, and which no evaluation reaches. So the translation needs no recursive binding, and the only recursion in a program is that of the two folds.

Folds and ranges. The iterator is the catamorphism of the chain of a natural number, a fold over any range of $\mathscr{R}$ is that catamorphism at the range's length with the range's position carried beside the state, and cata is the catamorphism of a term's inductive type. The paramorphism, which also hands each node itself to the algebra, is the iterator at $\mu F \times \sigma$ with the node rebuilt beside the result. A term type translates to μF, whose constructors are its node operations, and T-Node reads a construction as a term of that type, in_O of its fields, exactly where a term of type μF is expected, the argument of a cata or of an algebra, so that a term keeps one derivation. The kernel's ranges are library adapters, and any instance of $\mathscr{R}$ would do. n » chain(prior)

declarations	
template <> constexpr τ f<$\overline{\tau}$>($\overline{x}$) ... e	an instance
$\mapsto\ f_{\overline{\tau}} = \lambda\overline{x}.\,[\![e]\!]$	
template <$\overline{\alpha}$> requires $\overline{Q}$ ... $f\,(\overline{x})$ ... e	a generic function
$\mapsto\ f = \Lambda\overline{\alpha}.\,\lambda\overline{q : [\![Q]\!]}.\,\lambda\overline{x}.\,[\![e]\!]$	
the same with a descending self-call on $p \in \overline{x}$	
$\mapsto\ f = \Lambda\overline{\alpha}.\,\lambda\overline{q}.\,\lambda\overline{x}.\,(iter\,(toNat\,p + 1)\,F\,d)\,\overline{x}$	the one exception, $F = \lambda f.\,\lambda\overline{x}.\,[\![e]\!]$
struct O { $\overline{y}$; ... operator()($\overline{x}$) ... e ... };	a function object's call function
$\mapsto\ O_{\text{call}} = \lambda\overline{y}.\,\Lambda\overline{\beta}.\,\lambda\overline{q : [\![Q]\!]}.\,\lambda\overline{x}.\,[\![e]\!]$	
O standard $\mapsto\ O_{\text{call}} = \lambda().\,o$	the standard's function o as a constant
concept, = delete $\mapsto$ nothing	
terms	
$[\![f$<$\overline{\sigma}$>$(\overline{e})]\!] = f_i\,\overline{[\![\sigma]\!]}\,\overline{[\![\Delta \Vdash Q[\overline{\sigma}/\overline{\alpha}]]\!]}\,\overline{[\![e]\!]}$	T-Call, f_i the declaration it resolves to
$[\![f$<$\overline{\sigma}$>$(\overline{e})]\!] = f_{\overline{\sigma}}\,\overline{[\![e]\!]}$	T-Trait, resolved to a specialisation
$[\![\tau^S(e_r[\overline{e}/\overline{a}])]\!] = q_C.\pi.r\,\overline{[\![e]\!]}$	T-Req-Assumed, π the path through nested shapes
$[\![O$<$\overline{\tau}$>$\{\overline{e_y}\}]\!] = (\overline{[\![e_y]\!]},\ O_{\text{call}}\,\overline{[\![e_y]\!]})$	construction; a fieldless O has the empty tuple of fields
$[\![e(\overline{e'})]\!] = (\pi_2\,[\![e]\!])\,[\overline{[\![\sigma]\!]}]\,\overline{d}\,\overline{[\![e']\!]}$	T-Op, T-App: application
with $\overline{d} = \overline{[\![\Delta \Vdash Q[\overline{\tau}/\overline{\alpha}][\overline{\sigma}/\overline{\beta}]]\!]}$	the call operator's dictionaries
$[\![e.y_i]\!] = \pi_i\,(\pi_1\,[\![e]\!])$	a field, through the fields' pair
$[\![\tau(e)]\!] = conv_{\tau',\tau}\,[\![e]\!]$	a conversion, a constant of type $[\![\tau']\!] \to [\![\tau]\!]$, the identity's δ-step included at $\tau' = \tau$

Figure 7 The translation of declarations and terms; pairs, sums, boxes and conditionals are the target's own (Table 9). The right-hand column says which case a line is.

a step's callable

$[\![f \texttt{<}\overline{\sigma}\texttt{>}]\!]^\circ = f\ \overline{[\![\sigma]\!]}\ \overline{d}$ — a function name

$[\![e]\!]^\circ = (\pi_2\,[\![e]\!])\,[\overline{[\![\sigma]\!]}]\,\overline{d}$ — a function object, elaborated at the fold's types

with $\overline{d}$ the dictionaries of its constraints at $\overline{\sigma}$ — its dictionaries supplied

$[\![e]\!]^\circ = \lambda \overline{a}.\, q_C.\pi.r\,\overline{a}$ — a step typed by T-Req-Assumed: the requirement's field

recursion on a count, and on a term

$[\![\mathsf{fold_left}(r, a, f)]\!] =$

$\pi_2\,(iter\,[[\![R]\!] \times \sigma]\,(toNat\,|[\![r]\!]|)\,step\,([\![r]\!], [\![a]\!]))$ — the iterator, over the range's length read as a natural

where $step = \lambda p.\,(next\,(\pi_1 p),\ [\![f]\!]^\circ\,(\pi_2 p)\,(head\,(\pi_1 p)))$ — the position carried beside the state

with $|\cdot|, head, next$ the operations of R at a type variable R (T-Fold-R):

$toNat \circ q.size, q.head, q.next$ in their place — q the record of *IsRange*<R, τ> in Δ

$[\![\mathsf{cata}(alg, t)]\!] = \pi_2\,(fold_F\,[\mu F \times \sigma]\,\beta\,[\![t]\!])$ — a fold over a term

where $\beta = \lambda n'.\,(\mathrm{in}\,(F\ \pi_1\ n'),$

$[\![alg]\!]^\circ\,(\mathrm{in}\,(F\ \pi_1\ n'))\,(F\ \pi_2\ n'))$ — the node rebuilt beside its result

$[\![t]\!]_{\mu F} = node_{O\texttt{<}\overline{\tau}\texttt{>}}\,[\![t]\!]$ — T-Node: any term of node type read where μF is expected

where $node_O = \lambda o.\,\mathrm{in}_O\,(\overline{node_{O_l}\,(\pi_l\,(\pi_1\,o))}, \overline{\pi_y\,(\pi_1\,o)})$ — its legs, of node types O_l smaller than O, read the same way, its other fields as they are

Figure 8 The translation of a step's callable, of a fold and of a term's fold. A term is read into its inductive type where one is expected.

is the chain of a count under Figure 2's *prior*: head prior(n), tail prior(n) » chain(prior), empty where prior(n) is nothing, finite because the step descends (Section 2). transform(r, g) is Listing 3's r » g: head g(head r), tail transform(next r, g), length $|r|$. iota(i, j), with $i \leq j$, as reversed endpoints are undefined behaviour and not an empty range, has head i and tail iota(i + 1, j). Each instance R of $\mathscr{R}$ over τ contributes a state type $[\![R]\!]$, a constructor constant c_R, and $|\cdot| : [\![R]\!] \to [\![\text{size_t}]\!]$, $\mathit{head} : [\![R]\!] \to [\![\tau]\!]$ and $\mathit{next} : [\![R]\!] \to [\![R]\!]$, constants for a base range and terms over the parts for a range built on another. A range translates to $c_R\,\overline{[\![e]\!]}$ on its arguments, and size, head and next to those three. The state of n » chain(prior) is $[\![N]\!]$, that of transform(r, g) is $[\![R]\!] \times [\![g]\!]$, the arrow carried so that *head* applies it and *next* keeps it, and that of iota(i, j) is $\mathbb{N} \times \mathbb{N}$.

Boxes. A box translates to a unary product rather than to nothing, on purpose, make_shared to $((), [\![e]\!])$ and *e to $\pi_2\,[\![e]\!]$: a box's only observable is its content, as no get, use_count or == between boxes is a term, so which box holds it is erased, and the unit component keeps unboxing a step on both sides, which Lemma 5 needs.

Conventions. Table 9 is the fragment's convention on the standard library: each type and helper, adopted by opinion, has rules of its own in Appendices C and E.

System F	FJlt	C++	why not declared
$\tau \times \sigma$, (a, b), π_1, π_2 [79]	pair<τ, σ>, pair{a, b}, .first, .second	the same	to land on the target's products
$1 + \tau$, inl, inr, case	optional<τ>, $\iota_1\,e$, $\iota_2\,e$, case(s, f, g)	nullopt, optional{e}, s.transform(g).value_or(f), f the value for the empty one	an encoding has junk in the unused field
$\tau + \sigma$, inl, inr, case	variant<τ, σ>, $\iota_1\,e$, $\iota_2\,e$, case(s, f, g)	variant<τ, σ>(in_place_index<i-1>, e), a dispatch on the index, as the library's Copairing	the same
a unary product	shared_ptr<const τ>, make_shared, *	the same	a box has no identity to observe
a state $[\![R]\!]$, *iter*	R<τ>, fold_left(r, a, f), size, head, next	std::ranges::fold_left, ranges::size, *ranges::begin under a guard, the instance's own	it is the recursion
μF, its fold	nested nodes O<$\overline{\tau}$>, cata(alg, t)	the library's	it is the recursion

Table 9 The fragment's convention on the standard library. std::visit is the case whose two branches are one operation.

E Evaluation

Evaluation is small-step and call-by-value on closed terms, $e \longrightarrow e'$, with values and evaluation contexts

$$\begin{array}{rcl} v & ::= & c \mid f \mid O\texttt{<}\overline{\tau}\texttt{>}\{\overline{v}\} \mid \texttt{std::pair}\{v, v\} \mid \iota_1\, v \mid \iota_2\, v \mid \texttt{std::optional}\{v\} \mid \texttt{std::nullopt} \\ & \mid & \texttt{std::make_shared<const } \tau\texttt{>}(v) \\ E & ::= & [\,] \mid f\texttt{<}\overline{\sigma}\texttt{>}(\overline{v}, E, \overline{e}) \mid E(\overline{e}) \mid v(\overline{v}, E, \overline{e}) \mid \ominus E \mid E \oplus e \mid v \oplus E \mid \tau(E) \mid E\ \texttt{?}\ e : e \\ & \mid & \iota_i\, E \mid \mathsf{case}(E, e, e) \mid \mathsf{case}(v, E, e) \mid \mathsf{case}(v, v, E) \mid \ldots \end{array}$$

where the elided contexts are the remaining argument positions of pairs, optionals, boxes, projections, constructions, folds and the two branches of a case, each evaluated left to right. This is FJlt's own evaluation, the calculus's stand-in for `clang++` and *run* in Figure 1, which Appendix F shows agrees with *eval* across $[\![\cdot]\!]$. It is C++'s constant evaluation restricted to the grammar, extended by S-Unbox at run time, and Figure 9 gives the rules; $e \Downarrow v$ abbreviates $e \longrightarrow^* v$. The context grammar picks one redex per non-value term, so the relation is deterministic. A value is immutable and a reference to one denotes it, which is why a parameter written **const** τ & is the parameter $\tau\ x$.

F Theorems

The statements are in Section 5; the proofs are here as sketches, relative to the rules of Figures 4 to 9, whose omitted cases (constants, pairs, boxes, conditionals, the arithmetic of base types) are the textbook ones; what the sketches leave open is listed after them. Throughout, Σ is any signature whose carriers meet the conditions of Appendix A, used only as constants and total operations, and the translation $[\![\cdot]\!]$ is the one of Appendix D, into System F with products, sums, the carriers of Σ as constants and the iterators. Its carriers are non-empty, since Rule 1 (Substitution) makes them default constructible, so the usual equational calculus stays sound and complete in the many-sorted case [38]. A context translates pointwise: $[\![\Delta]\!]$ has a type variable for each $\alpha \in \Delta$ and a record variable $q_Q : [\![Q]\!]$ for each assumed constraint $Q \in \Delta$, and $[\![\Gamma]\!] = \overline{x : [\![\tau]\!]}$. The target's iterators reduce by $\mathit{iter}\,(k{+}1)\,s\,z \longrightarrow \mathit{iter}\,k\,s\,(s\,z)$, $\mathit{iter}\,0\,s\,z \longrightarrow z$ and $\mathit{fold}_F\,\beta\,(\mathrm{in}\,n) \longrightarrow \beta\,(F(\mathit{fold}_F\,\beta)\,n)$. Reduction in the target is full β-reduction, any redex anywhere, together with projections, case, the δ-rules of the constants, the unfolding of a defined name to its body and the two iterator rules; no evaluation order is imposed. System F with products, sums and positive inductive types with their iterators is strongly normalising and confluent, and its iterative fragment embeds into System F by an encoding that preserves reduction [67]; for first-order rewrite rules on algebraic constants, Breazu-Tannen and Gallier show that adding them to System F conserves strong normalisation and confluence [12, 13]. These results cover the parts separately, and the translation assumes their combination, which the same method gives: *iter* in the accumulator form above is the standard iterator at $\sigma \to \sigma$, each of its steps at least one step of that; the defined names are acyclic, by the declaration order, and expand away; the δ-rules are ground, a constant out of constants, and preserve reducibility; and confluence then follows by

(S-Call) $f\langle\overline{\sigma}\rangle(\overline{v}) \longrightarrow e[\overline{\sigma}/\overline{\alpha}][\overline{v}/\overline{x}]$
where $f\langle\overline{\sigma}\rangle$ resolves to a declaration with parameters $\overline{x}$ and body e

(S-Op) $O\langle\overline{\tau}\rangle\{\overline{v_y}\}(\overline{v}) \longrightarrow e[\overline{\tau}/\overline{\alpha}][\overline{\sigma}/\overline{\beta}][\overline{v_y}/\overline{y}][\overline{v}/\overline{x}]$ for a declared operation with body e, $\overline{\sigma}$ deduced

(S-Std) $O\langle\overline{\tau}\rangle\{\}(\overline{c}) \longrightarrow c$ for a standard function object, c the value the standard assigns

(S-Prim) $\ominus c \longrightarrow c' \qquad c_1 \oplus c_2 \longrightarrow c$ the base type's operator, total by the choice of $\ominus, \oplus$

(S-Conv) $\tau(c) \longrightarrow c'$ the conversion between distinct carriers the standard or the declaration assigns $\qquad \tau(v) \longrightarrow v$ at v's

(S-If) $\mathsf{true}\ ?\ e_1 : e_2 \longrightarrow e_1 \qquad \mathsf{false}\ ?\ e_1 : e_2 \longrightarrow e_2$

(S-Proj) $\mathsf{std::pair}\{v_1, v_2\}.\mathsf{first} \longrightarrow v_1 \qquad \mathsf{std::pair}\{v_1, v_2\}.\mathsf{second} \longrightarrow v_2 \qquad O\langle\overline{\tau}\rangle\{\overline{v}\}.y_i \longrightarrow v_i$

(S-Unbox) $\mathsf{*make_shared}\langle\mathsf{const}\ \tau\rangle(v) \longrightarrow v$

(S-Case) $\mathsf{case}(\iota_1\, v, v_f, v_g) \longrightarrow v_f(v) \qquad \mathsf{case}(\iota_2\, v, v_f, v_g) \longrightarrow v_g(v)$

(S-Range) $\mathsf{size}(r) \longrightarrow |r| \qquad \mathsf{head}(r) \longrightarrow \mathit{head}\, r \text{ if } |r| > 0,\ \mathit{default}_\tau \text{ if } |r| = 0 \qquad \mathsf{next}(r) \longrightarrow \mathit{next}\, r \text{ if } |r| > 0,\ r \text{ if } |r| = 0$

(S-Fold) $\mathsf{fold_left}(r, v_a, v_f) \longrightarrow \mathsf{fold_left}(\mathit{next}\, r, v_f(v_a, \mathit{head}\, r), v_f) \quad \text{if } |r| > 0$
$\mathsf{fold_left}(r, v_a, v_f) \longrightarrow v_a \quad \text{if } |r| = 0$

(S-Cata) $\mathsf{cata}(v_{alg}, n) \longrightarrow v_{alg}(n, \mathsf{cata}(v_{alg}, l_1), \ldots, \mathsf{cata}(v_{alg}, l_k))$
for a node n with legs $l_1 \ldots l_k$

(S-Ctx) $E[e] \longrightarrow E[e'] \quad \text{if } e \longrightarrow e'$

Figure 9 Evaluation. S-Call covers specialisations and generic functions alike, since resolution has already chosen the body; for a generic function the explicit type arguments are substituted first. S-Range is total: on the empty range the head is $\mathit{default}_\tau$ and the tail the range itself, values S-Fold never reaches, since it runs $|r|$ times. The default value of a type is given by its structure, not by T{}: a carrier's own default, a pair of defaults, the empty optional, the first injection of a default, a fieldless function object, and for a box a box of its content's default, so that no term of the kernel holds a null box. S-Fold consumes the head of the range, so a fold takes as many steps as the range has elements. S-Cata unfolds one node, passing the node itself beside its legs' folds; a leaf has no legs and the algebra is applied to it alone. S-Prim and S-Std are total because the admitted operations are, which is what Lemma 4 needs. That totality, the condition of Appendix A, is a reading: neither **constexpr** nor a successful evaluation shows an operation defined on every input, and it keeps **int**'s overflow and division out. On unsigned carriers the operations are addition, subtraction, multiplication and the bitwise operations, which wrap, the comparisons, and division, remainder and the shifts by a literal the carrier can take (a nonzero divisor, a shift below the width); by a variable, those three are partial and not admitted. On **bool** they are the logical operations. Σ is not a list; the listings also use std::string with concatenation and the comparisons, allocation failure aside, and finite enumerations such as Ternary. S-Unbox is evaluation at run time only: std::shared_ptr is not **constexpr** in C++23, so a constant evaluator rejects a term that builds a box. A law or a **static_assert** premise therefore contains no make_shared, the condition under which every FJlt program is a Jlt program. C++26 retires that condition [50].

Newman's lemma, there being no critical pairs. That is a sketch, not a proof, and the one-constructor-at-a-time iterators are what make Theorem 2 exact where a Church encoding, unfolding all at once, would not be. Three facts about the target are used without proof: it is strongly normalising and confluent, as above, and substitution of types for type variables preserves typing.

Lemma 1 (Dictionaries are well typed)**.** *If* $\Delta \Vdash Q$*, then* $[\![\Delta]\!]; \varnothing \vdash [\![\Delta \Vdash Q]\!] : [\![Q]\!]$.

Proof. By induction on the entailment derivation, mutually with Theorem 1, the induction being on the size of the derivation. E-Assume: the dictionary is the variable q_Q, which $[\![\Delta]\!]$ types at $[\![Q]\!]$. E-Shape: $Q = C\langle\overline{\tau}\rangle$, the dictionary is the record $\{\overline{r_i = [\![\Delta \Vdash A_i[\overline{\tau}/\overline{\alpha}]]\!]}\}$, each field has type $[\![A_i]\!][\overline{\tau}/\overline{\alpha}]$ by the induction hypothesis, and that is the field type of $[\![C\langle\overline{\tau}\rangle]\!]$ by definition. E-Law and E-Law-Assumed: the dictionary is *true* : *Bool*. E-Not: the dictionary is $() : 1$. E-Op: the premise types e at τ' under $\overline{x : \tau_x}$, so by Theorem 1 on a smaller derivation $[\![e]\!] : [\![\tau']\!]$ under $\overline{x : [\![\tau_x]\!]}$, and $\lambda\overline{x}.[\![e]\!] : \overline{[\![\tau_x]\!]} \to [\![\tau']\!]$; for std::same_as there is no conversion, and for std::convertible_to the field applies the conversion constant $conv_{\tau',\tau} : [\![\tau']\!] \to [\![\tau]\!]$ (T-Conv), so the field has type $\overline{[\![\tau_x]\!]} \to [\![\tau]\!]$ in both cases. □

Proof of Theorem 1, type preservation. By induction on the typing derivation, one case per rule. T-Var: $[\![x]\!] = x$, and $[\![\Gamma]\!]$ types it at $[\![\tau]\!]$. T-Construct: $[\![O\langle\overline{\tau}\rangle\{\overline{e_y}\}]\!] = (\overline{[\![e_y]\!]}, O_{\text{call}}\,\overline{[\![e_y]\!]})$; the fields have their types by the induction hypothesis, and $O_{\text{call}}\,\overline{[\![e_y]\!]} : \forall\overline{\beta}.\overline{[\![Q]\!]} \to \overline{[\![\tau_x]\!]} \to [\![\tau]\!]$ by the induction hypothesis on the operation's body, typed under $\overline{\beta}, \overline{Q}$ as a generic function's is (for a standard function object, by the assumption on the standard's function), so the pair has type $[\![O\langle\overline{\tau}\rangle]\!]$. T-Op and T-App: $\pi_2\,[\![e]\!]$ has the call function's type, is instantiated at the deduced $\overline{\sigma}$, receives the dictionaries $\overline{[\![\Delta \Vdash Q[\overline{\sigma}/\overline{\beta}]]\!]}$, of the parameter types by Lemma 1, and the arguments, of theirs by the induction hypothesis; T-Field: $\pi_i\,(\pi_1\,[\![e]\!])$ has the field's type; T-Conv: the conversion constant has type $[\![\tau']\!] \to [\![\tau]\!]$. T-Inj and T-Case: an injection translates to the target's, and a case to the target's case with each branch elaborated as a callable at its summand, of type σ by the induction hypothesis. T-Call: f was declared by D-Generic, so $f = \Lambda\overline{\alpha}.\lambda\overline{q : [\![Q]\!]}.\lambda\overline{x}.[\![e_f]\!]$ has type $\forall\overline{\alpha}.\overline{[\![Q]\!]} \to \overline{[\![\tau_x]\!]} \to [\![\tau]\!]$ by the induction hypothesis on that declaration; applying it to $\overline{[\![\sigma]\!]}$ gives, by Lemma 2, $\overline{[\![Q[\overline{\sigma}/\overline{\alpha}]]\!]} \to \overline{[\![\tau_x[\overline{\sigma}/\overline{\alpha}]]\!]} \to [\![\tau[\overline{\sigma}/\overline{\alpha}]]\!]$; the records have the parameter types by Lemma 1, and the arguments theirs by the induction hypothesis. T-Trait: the call resolved to a specialisation declared by D-Instance, so $f_{\overline{\sigma}} = \lambda\overline{x}.[\![e]\!] : \overline{[\![\tau_x]\!]} \to [\![\tau]\!]$ by the induction hypothesis on that declaration, and the arguments have the right types; if it resolved to a generic function with an entailed context, the case is T-Call with the context's dictionaries. T-Req-Assumed: $C\langle\overline{\tau}\rangle \in \Delta$, so $q_C : [\![C\langle\overline{\tau}\rangle]\!]$ is in $[\![\Delta]\!]$, the projection $q_C.r$ has the requirement's field type, the conversion included, $\overline{[\![\tau_a]\!]} \to [\![\tau]\!]$, and the arguments have the parameters' types by the induction hypothesis. T-Fold, for a range of instance R over τ: $[\![r]\!] = c_R\,\overline{[\![e]\!]} : [\![R]\!]$ by the induction hypothesis on the arguments and the constructor's type, $iter[[\![R]\!] \times \sigma] : \mathbb{N} \to (([\![R]\!] \times \sigma) \to [\![R]\!] \times \sigma) \to [\![R]\!] \times \sigma \to [\![R]\!] \times \sigma$, $toNat\,|[\![r]\!]| : \mathbb{N}$, and the step has the pair type since $next : [\![R]\!] \to [\![R]\!]$, $head : [\![R]\!] \to [\![\tau]\!]$ and the callable, elaborated at σ and τ with its type arguments applied and its dictionaries supplied by Lemma 1, has type $\sigma \to \tau \to \sigma$

(curried in the target); $(\llbracket r \rrbracket, \llbracket a \rrbracket)$ has the pair type, and π_2 of the result has type σ. T-Fold-R: the same with *q.size*, *q.head* and *q.next* for the operations, typed as the fields of $q : \llbracket \mathit{IsRange}\langle R, \tau \rangle \rrbracket$ in $\llbracket \Delta \rrbracket$, the coercion to $\mathbb{N}$ applied to the length. T-Range-Op: $|\llbracket r \rrbracket|$, $\mathit{head}\llbracket r \rrbracket$ and $\mathit{next}\llbracket r \rrbracket$ have the operations' types, $\llbracket \text{size_t} \rrbracket$, $\llbracket \tau \rrbracket$ and $\llbracket R \rrbracket$. T-Node: node_O has type $\llbracket O\langle \overline{\tau} \rangle \rrbracket \to \mu F$, by induction on the term type, whose legs are smaller, and it applies to $\llbracket t \rrbracket$ of the node's type by the induction hypothesis. T-Cata: $\mathit{fold}_F\,[\mu F \times \sigma] : (F(\mu F \times \sigma) \to \mu F \times \sigma) \to \mu F \to \mu F \times \sigma$; β has the algebra's type since $\mathrm{in}(F\,\pi_1\,n') : \mu F$ rebuilds the node, $F\,\pi_2\,n'$ holds the legs' results of type σ, and $\llbracket alg \rrbracket^\circ$, elaborated at μF and σ, has type $\mu F \times \sigma^k \to \sigma$ by the induction hypothesis; $\llbracket t \rrbracket : \mu F$ by T-Node, and π_2 of the result has type σ. D-Generic: the body is typed under $\overline{\alpha}, \overline{Q}$ and $\overline{x : \tau_x}$, so by the induction hypothesis $\llbracket e \rrbracket : \llbracket \tau \rrbracket$ under type variables $\overline{\alpha}$, record variables $\overline{q : \llbracket Q \rrbracket}$ and $\overline{x : \llbracket \tau_x \rrbracket}$, and the abstraction has the stated polymorphic type; this is the case that uses T-Req-Assumed, since the body reaches an operation on a constrained type only through Δ. D-Op: as D-Generic, with the fields among the variables. D-Instance: the same with an empty Δ; the declaration order makes the induction on declarations well founded, a self-call of the one exception being typed by the declared signature of f, at the enclosing type variables and dictionaries, so that the binder of F has the monomorphic type $\overline{\llbracket \tau_x \rrbracket} \to \llbracket \tau \rrbracket$ at which the iterator runs, and its body checked once. The omitted rules are the textbook cases for pairs, sums and conditionals. □

Lemma 2 (Translation commutes with substitution). *(i)* $\llbracket \tau[\overline{\sigma}/\overline{\alpha}] \rrbracket = \llbracket \tau \rrbracket[\overline{\llbracket \sigma \rrbracket}/\overline{\alpha}]$. *(ii) For a term typed at concrete types,* $\llbracket e[\overline{v}/\overline{x}] \rrbracket = \llbracket e \rrbracket[\overline{\llbracket v \rrbracket}/\overline{x}]$. *(iii) If* $\Delta \Vdash Q$ *and* $\varnothing \Vdash Q[\overline{\sigma}/\overline{\alpha}]$, *then* $\llbracket \Delta \Vdash Q \rrbracket[\overline{\llbracket \sigma \rrbracket}/\overline{\alpha}][\overline{d}/\overline{q}] \longrightarrow^* \llbracket \varnothing \Vdash Q[\overline{\sigma}/\overline{\alpha}] \rrbracket$, *where* $\overline{d}$ *are the dictionaries of* Δ*'s constraints at* $\overline{\sigma}$. *(iv) For a term* e *typed under* Δ, $\llbracket e \rrbracket[\overline{\llbracket \sigma \rrbracket}/\overline{\alpha}][\overline{d}/\overline{q}] \longrightarrow^* \llbracket e[\overline{\sigma}/\overline{\alpha}] \rrbracket$, *the right side typed at the concrete types.*

Proof. (i) By induction on τ: every row of the type translation is structural in the subterms. (ii) By induction on e: every row of the term translation is structural in the subterms, and a variable translates to itself. (iii) and (iv) together, by induction on the entailment and typing derivations, after Lemma 3, which they use and which uses only (ii). E-Assume: the dictionary is q, and the substitution puts d, the dictionary built at $\overline{\sigma}$, in its place. E-Shape: the record is rebuilt field by field, each by the induction hypothesis. E-Op: the field is $\lambda\overline{a}.\llbracket \tau^S(e_r) \rrbracket$, and e_r is typed under Δ, so (iv) reduces it to the field E-Op builds at $\overline{\sigma}$. E-Law, E-Law-Assumed and E-Not: the same constant on both sides. (iv): a T-Req-Assumed site, $q.r\ \ldots$, has become $d.r\ \ldots$ and reduces by Lemma 3 to the resolved instance's translation; the dictionaries of a T-Call or T-App site, and the record of a T-Fold-R site, reduce by (iii); every other row is structural in its subterms. □

Lemma 3 (Dictionaries agree with resolution). *If* $\varnothing \Vdash C\langle\overline{\sigma}\rangle$ *with* $\overline{\sigma}$ *concrete and* `{ e_r } -> S<τ>` *is a requirement of* C *with parameters* $\overline{a}$, *then*

$$\llbracket \varnothing \Vdash C\langle\overline{\sigma}\rangle \rrbracket.r = \lambda\overline{a}.\llbracket \tau^S(e_r[\overline{\sigma}/\overline{\alpha}]) \rrbracket,$$

in which every call has been resolved; hence for any terms $\overline{e}$,

$$(\llbracket \varnothing \Vdash C\langle\overline{\sigma}\rangle \rrbracket.r)\,\overline{\llbracket e \rrbracket} \longrightarrow^+ \llbracket \tau^S(e_r[\overline{\sigma}/\overline{\alpha}][\overline{e}/\overline{a}]) \rrbracket,$$

by a projection and β-reductions, one per parameter.

Proof. The field is built by E-Op as $\lambda\overline{a}.[\![\tau^S(e_r[\overline{\sigma}/\overline{\alpha}])]\!]$, where the instantiated expression is typed at concrete types, so its calls are T-Trait or T-Call and translate to resolved terms; a projection of the constructed record followed by β-reduction on $\overline{[\![e]\!]}$, which full β-reduction permits whether or not the arguments are values, gives the stated term, by Lemma 2 (ii). □

Lemma 4 (Progress)**.** *A closed well-typed term is a value or takes a step.*

Proof. By induction on the typing derivation. If some proper subterm in an evaluation position is not a value, the leftmost such subterm steps by the induction hypothesis, and S-Ctx lifts the step. Otherwise the term is a value, or a redex of one of the following shapes, each of which steps. A call $f\langle\overline{\sigma}\rangle(\overline{v})$ typed by T-Call or T-Trait: the premise says the call resolves (for T-Call, the constraints are entailed, so resolution finds the generic function or, through its context, a specialisation), so S-Call applies; T-Req-Assumed cannot occur in a closed term, since Δ is empty. An application $O\langle\overline{\tau}\rangle\{\overline{v_y}\}(\overline{v})$: S-Op for a declared operation, S-Std for a standard one, whose operations are total by the choice of O. A primitive $\ominus c$ or $c_1 \oplus c_2$: S-Prim, total by the choice of $\ominus$ and $\oplus$. A conversion $\tau(c)$: S-Conv, total by the condition on conversions. A range's operation on a value of a range: S-Range, total by its extension to the empty range. A conditional on a constant of type **bool**, a case on a value of a sum, which by T-Case and canonical forms is an injection, a projection of a pair or of a constructed object, an unbox of a box, a fold whose count is a constant, and a cata on a node: S-If, S-Case, S-Proj, S-Unbox, S-Fold, S-Cata, with S-Fold's two cases covering every constant and S-Cata covering every node, leaves included. Preservation along a step, that $e \longrightarrow e'$ keeps the type, is the usual substitution argument, the conversion written in the kernel term keeping a requirement's use at its declared type after S-Call, and it is used below as source subject reduction. □

Three reductions of the target run ahead of the source: the fuel of a descending self-call, the count of a fold and the reading of a node into its constructor. Write $t \sqsupseteq e$ when t is $[\![e]\!]$ with some occurrences of the name f of a function of the one exception, applied to an argument whose value is w, replaced by $F^m\,d$ with $m \geq w+1$, some counts $toNat\,|[\![r]\!]|$ replaced by their numerals, and some readings $node_O\,[\![v]\!]$ of a node that is already a value replaced by their reducts, so that no pending source step is erased or anticipated; in particular $[\![e]\!] \sqsupseteq e$, and $t \sqsupseteq c$ only for $t = c$, a constant holding neither a name nor a count.

Lemma 5 (Forward simulation)**.** *If $t \sqsupseteq e$ and $e \longrightarrow e'$ then $t \longrightarrow^+ t'$ for some $t' \sqsupseteq e'$, and a value of base type translates to a constant.*

Proof. Values: a constant translates to a constant; a function object with fields translates to a pair whose second component, O_{call} applied to the fields, may still reduce, which is why the claim is made for base types. Steps, by cases on the rule; by S-Ctx and the fact that the translation of a context is a context with one or more copies of the hole, each of which steps, it suffices to treat the redex. S-Call to a specialisation: $f_{\overline{\sigma}}\,\overline{[\![v]\!]}$

β-reduces to $[\![e_f]\!][\overline{[\![v]\!]}/\overline{x}] = [\![e_f[\overline{v}/\overline{x}]]\!]$. S-Call to a generic function: $f\ \overline{[\![\sigma]\!]}\ \overline{d}\ \overline{[\![v]\!]}$ reduces by type application and β-reduction to $[\![e_f]\!]$ with $\overline{[\![\sigma]\!]}$ for $\overline{\alpha}$, $\overline{d}$ for $\overline{q}$ and $\overline{[\![v]\!]}$ for $\overline{x}$; by Lemma 2 (iv), with $\overline{d}$ the records E-Shape built for the closed call, the result reduces further to $[\![e_f[\overline{\sigma}/\overline{\alpha}][\overline{v}/\overline{x}]]\!]$, the term the source stepped to, whose requirement uses are typed by T-Trait, under T-Conv where the requirement converts, and whose folds at the former type variable are T-Fold's at the concrete instance. S-Call at the self-call of the one exception: where the source calls $f(w)$, t carries $F^m\,d$ with $m \geq w+1$, the first call $f\ v$ having unfolded to $(iter\,(v{+}1)\,F\,d)\,v \longrightarrow^+ F^{v+1}\,d\,v$; the β-steps of F's abstractions give the body with $F^{m-1}\,d$ at its self-calls, and $m-1 \geq w \geq w'+1$ for the argument $w' < w$ of each, so the result is $\sqsupseteq$ the source's reduct, and d is never reached. S-Op: $(\pi_2\,(\overline{[\![v_y]\!]}, O_{\text{call}}\,\overline{[\![v_y]\!]}))\,[\overline{\sigma}]\,\overline{d}\,\overline{[\![v]\!]}$ reduces by projection, type application and β-reduction to $[\![e_O]\!]$ with fields, dictionaries and arguments substituted, which is $[\![e']\!]$ by the same argument as for a generic call. S-Std, S-Prim and S-Conv: a projection where there is one, then a δ-step to the constant the standard or the declaration assigns, by the assumption that the primitives' translations compute the standard's functions on constants. S-If, S-Case, S-Proj and S-Unbox: a case reduction, a case reduction to the translation of the selected application, whose evaluation is the next step's, a projection, a projection; the box is a unary product rather than nothing so that unboxing is a step on both sides and the simulation stays strict. S-Fold with $|r| = k+1$:

$$iter\,(k{+}1)\,step\,([\![r]\!], [\![a]\!]) \longrightarrow iter\,k\,step\,(step\,([\![r]\!], [\![a]\!])) \longrightarrow^+ iter\,k\,step\,([\![next\,r]\!], [\![f]\!]^\circ\,[\![a]\!]\,[\![head\,r]\!]),$$

the operations *next* and *head* computing the range's, and the last term is $\sqsupseteq$ the source's reduct, the fold over *next r* from $f\,(a, head\,r)$, whose translation differs from it only by the count $toNat\,|[\![next\,r]\!]|$, which reduces to k; the empty case is one step of the iterator followed by a projection. S-Cata: $fold_F\,\beta\,(\text{in}\,n) \longrightarrow \beta\,(F(fold_F\,\beta)\,n)$ is one step of the term's iterator; β then rebuilds the node from the first components, which reduces to $[\![n]\!]_{\mu F}$ since each leg's pair has that leg, read at μF, as its first component, and applies $[\![alg]\!]^\circ$ to it and to the second components $\pi_2\,(fold_F\,\beta\,[\![l_i]\!])$, which are $[\![\mathsf{cata}(alg, l_i)]\!]$; after those projections the term is $\sqsupseteq$ the source's reduct, the node's reading there already reduced. □

Proof of Corollary 1, termination. An infinite reduction $e \longrightarrow e_1 \longrightarrow \cdots$ would give, by Lemma 5 from $[\![e]\!] \sqsupseteq e$, an infinite reduction $[\![e]\!] \longrightarrow^+ t_1 \longrightarrow^+ \cdots$ with $t_i \sqsupseteq e_i$ in the target, which is strongly normalising. □

Proof of Corollary 3, type soundness. Let e be closed and well-typed at a base type. By Lemma 4 and preservation along steps, every term in the reduction sequence from e is closed and well-typed, so each is a value or steps; by Corollary 1 the sequence is finite; so it ends in a value v, and at a base type a value is a constant. □

Proof of Theorem 2, simulation. The first two claims are Lemma 5, the first up to $\sqsupseteq$. For the converse at a base type: if $e \Downarrow v$ then $[\![e]\!] \longrightarrow^* t \sqsupseteq v$ by Lemma 5 and induction on the sequence, and $t = [\![v]\!]$, a constant. Conversely, if $[\![e]\!] \longrightarrow^* c$ for a constant c: by Corollary 3 $e \Downarrow v$ for some v, so $[\![e]\!] \longrightarrow^* [\![v]\!]$ by the half just proved, and $c = [\![v]\!]$

since the target is confluent and both are normal forms, the translation being injective on constants. □

Proof of Theorem 3, coherence. Resolution is a function by well-formedness: at concrete types one specialisation at a head and a unique most specific generic function among the entailed candidates, and at a type variable only a call its context decides, so that the declaration a body's call resolves to is the same at every instantiation of the body and the dictionary built at the type variable is the one built after substitution; contexts are well-founded, so the recursion through contexts terminates with one answer. E-Not applies only at concrete types, where entailment is decided, and only to a negation not already assumed, as E-Shape applies only to a shape not in Δ, so it adds no choice. The entailment rules are syntax-directed, exactly one rule per constraint: E-Assume for a constraint in Δ, E-Shape for a shape not in Δ, by its side condition, E-Op for an operation, and E-Law or E-Law-Assumed for a law, which build the same *true*; so two derivations of $\Delta \Vdash Q$ have the same shape, and by induction on that shape they build the same dictionary: the same variable, the same record of fields by the induction hypothesis, the same *true*, the same $\lambda\overline{x}.[\![e]\!]$ in which every trait call resolved the same way. T-Req-Assumed and E-Law-Assumed are deterministic as well, taking the leftmost assumed shape and its first matching requirement in declaration order, nested shapes included, so the translation of a body is a function of the body. T-Req-Assumed, with T-Conv where its requirement converts, is tried before T-Call and T-Trait, T-Node applies exactly where a term of type μF is expected, and T-Call names the declaration resolution selects, so a call has one derivation even where several generic functions of one name have entailed contexts. Hence $[\![\Delta \Vdash Q]\!]$ depends only on Δ and Q, the records in T-Call are determined by the program, and the translation is a function of it. □

Proof of Corollary 2, parametricity. By Theorem 1, a generic FJlt function translates to a closed System F term of type $\forall\overline{\alpha}.\overline{[\![Q]\!]} \to \overline{[\![\tau_x]\!]} \to [\![\tau]\!]$; Reynolds' abstraction theorem [88, 112] gives its free theorem, in which the record parameters are ordinary term parameters, so the theorem relates behaviour at two instantiations through the relation between their records. The target's constants are interpreted by the identity relation at their base types and its iterators by the least relations closed under in, so the abstraction theorem extends to it, and the term is closed once the defined names are expanded. □

The dependencies run one way: coherence, which makes the translation a function of the term; preservation with the lemma on dictionaries; the substitution lemma, whose third and fourth parts rest on the lemma on resolution; source subject reduction; progress; forward simulation, which uses coherence to translate the reduct; termination from forward simulation and the target's strong normalisation; soundness from progress and termination; the converse of simulation from soundness; parametricity on its own. What the sketches leave open: they take fold_left over the ranges of $\mathscr{R}$ as a primitive with the evaluation rule of Figure 9, trusting the standard library for iota (Section 7); the conditions of T-Range are semantic, so termination is in part

assumed in the typing; the omitted cases are the textbook ones; and the inductions are on paper, with no mechanisation. Their mechanisation is future work.

G The compiler's own functional language

Instantiation is partial evaluation of the dictionary-passing term at a static type, $[\![concat]\!]$`[size_t]`d with its type and record arguments applied and β-reduced before the program runs, so the two agree on every closed term: Theorem 2 at the binding time where the records are constants, nothing of a record surviving to run time, and parametricity, a property of the term before that evaluation, holds of every instantiation as an application of it. For Go, Ellis et al. [31] prove the dictionary-passing translation correct beside monomorphisation, which Griesemer et al. [40] proved; C++ only ever does the second, and the first carries the theorems. What that evaluation leaves is a closed first-order term over the base types: the type abstractions are gone with their arguments, the record parameters are gone with the records the compiler resolved, every trait call is a call to one specialisation, and the only recursion left is *iter* and $fold_F$. The compiler then lowers that term to its intermediate representation, and the representation is in static single assignment form [25]: every variable is assigned exactly once, and where two paths of control join, a ϕ-node names the value that arrives along each.

SSA is a functional language. Kelsey showed it to correspond to continuation-passing style [51], and Appel put the correspondence as a slogan [4]: a basic block is a local function, its ϕ-nodes are that function's parameters, a branch is a call that passes the arguments, and dominance is lexical scope. So a Jlt function is translated twice into a functional language, by Appendix D into System F with records and iterators, and by the compiler into first-order functions with tail calls, with instantiation between the two; Rules 1 and 3 are what makes the source already look like the second.

Listing 7 is a search under a budget as the compiler leaves it before it unrolls the loop: `first<Over1000>`, a fold of one step over the first `budget` naturals, the state a candidate and whether it has the property, frozen once it has, read out as an optional, here behind a non-template wrapper. The fold over `iota(0, budget)` with that step is one loop, and its header has three ϕ-nodes: the two components of the state, the candidate and whether it has been found, and the position in the range. Those are the iterator's state in Figure 7, the pair $([\![r]\!], [\![a]\!])$ with $[\![a]\!]$ itself a pair, and the count $|r|$, here $[\![j]\!] - [\![i]\!]$, is the comparison that ends the loop. The exit block's two ϕ-nodes are the fold's result, read out into the optional. Nothing else survives: not the predicate's concept, not the record its constraint became, not the range object, not the pair; the predicate is the multiplication and the comparison in the loop body, and the budget is the register the wrapper was called with.

Into the vector unit. A vector register is a direct product of lanes, and its operations act pointwise, so a computation fits it when two things hold: no position depends on another, and the values are contiguous and small, under operations the unit has. Either can fail alone, as Table 10 shows. The table of Listing 5, Knuth, Morris and

```
define { i64, i8 } @first_rt(i64 %0) {
  %2 = icmp ult i64 %0, 2
  br i1 %2, label %15, label %3

3:                                        ; loop header
  %4 = phi i8 [ %12, %3 ], [ 0, %1 ]
  %5 = phi i64 [ %10, %3 ], [ 0, %1 ]
  %6 = phi i64 [ %13, %3 ], [ 1, %1 ]
  %7 = trunc i8 %4 to i1
  %8 = mul i64 %6, %6
  %9 = icmp ugt i64 %8, 1000
  %10 = select i1 %7, i64 %5, i64 %6
  %11 = zext i1 %9 to i8
  %12 = or i8 %4, %11
  %13 = add i64 %6, 1
  %14 = icmp eq i64 %13, %0
  br i1 %14, label %15, label %3

15:                                       ; exit
  %16 = phi i64 [ 0, %1 ], [ %10, %3 ]
  %17 = phi i8 [ 0, %1 ], [ %12, %3 ]
  %18 = trunc i8 %17 to i1
  %19 = select i1 %18, i64 %16, i64 0
  %20 = insertvalue { i64, i8 } poison, i64 %19, 0
  %21 = insertvalue { i64, i8 } %20, i8 %17, 1
  ret { i64, i8 } %21
}
```

Listing 7 first<Over1000>(budget) in LLVM IR, Clang 23 at -std=c++23 -O2 with unrolling and vectorisation off, attributes and metadata stripped; Appendix G reads it.

Pratt's and for many words Aho and Corasick's [2], is generic over the alphabet and fails the first, as each state needs the last. fixed_by(~rev) gives the first: the ranks embed the alphabet with its hole in a byte and carry the reflection to $3-r$, so a pair is a sum compared with three. count has an overload keyed on the section's type: a block goes through in stages, ranks, a pass per pair and a sum, each a fold small enough to be seen as a loop. It is an exhibit for this compiler and these flags: a specialisation is keyed on a law or a data shape, never on a particular problem or compiler, and it guarantees the answer, while vectorising stays the compiler's choice. Generating such code from algebra is old [33, 83], as is separating an algorithm from its schedule [85] and matching many patterns in lanes [114]; what is shown here is that standard C++23 gets there from laws its types already carry. The times, the disassembly and what else was tried are in the repository's src/bench, and make bench reproduces them.

This is a reason, not a theorem: the compiler is not among the hypotheses, its lowering is trusted, as every C++ program trusts it, and the reading of SSA as iterator terms is Appel's, which this paper uses and does not prove. Since both targets make the loop's state a parameter, the sentence of Section 6, that the discipline costs nothing

Counting the sites of pali6	independent	batchable	ns a letter
each window asked, as defined	yes	no	20
the table	no	yes	2
the same table, a hand-written loop	no	yes	2
the law, in one predicate, not inlined	yes	no	10
the law, a set of bases a byte	yes	yes, dearly	4
the law, in the library	yes	yes	**0.4**
the same, from Python	yes	yes	0.4
Python's re, a lookahead of 64 alternatives			200

Table 10 The two conditions, one failing at a time. One core of a shared AMD EPYC 9V74, Clang 23 at -O2, baseline SSE2, 10^8 letters, to one significant digit: benchmarks are fickle, and the orders of magnitude are the result. Past the cache, on 3.1 billion letters, the law runs at 3 GB/s against 30 GB/s for a plain read of the same memory, so the scan is bound by the core. A text drawn and not stored, by the table, keeps peak memory flat from 10^7 to 10^{10} letters; the law's own storage is a fixed block.

at run time, has a structural ground: a Jlt fold names in the source the parameter the compiler would otherwise recover from assignments. One seam shows here too: SSA's values are the scalars in registers, and a value that lives in memory, a boxed value in particular, is read and written through a pointer that no ϕ-node carries, which is the same line between a value and a cell that Rule 1 draws.

References

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

[2] Alfred V. Aho and Margaret J. Corasick. "Efficient String Matching: An Aid to Bibliographic Search". In: *Communications of the ACM* 18.6 (1975), pages 333–340. DOI: 10.1145/360825.360855.

[3] Noga Alon, Yossi Matias, and Mario Szegedy. "The Space Complexity of Approximating the Frequency Moments". In: *Journal of Computer and System Sciences* 58.1 (1999), pages 137–147. DOI: 10.1006/jcss.1997.1545.

[4] Andrew W. Appel. "SSA is functional programming". In: *ACM SIGPLAN Notices* 33.4 (1998), pages 17–20. DOI: 10.1145/278283.278285.

[5] Apple Inc. and the Swift project. *The Swift Programming Language: Generics*. Section Type Constraints. URL: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/generics/ (visited on 2026-10-24).

[6] John Backus. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". In: *Communications of the ACM* 21.8 (1978), pages 613–641. DOI: 10.1145/359576.359579.

[7] Aaron Ballman. *SEI CERT C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems in C++*. Software Engineering Institute, Carnegie Mellon University, 2016.

[8] Jean-Philippe Bernardy, Patrik Jansson, Marcin Zalewski, and Sibylle Schupp. "Generic programming with C++ concepts and Haskell type classes: a comparison". In: *Journal of Functional Programming* 20.3–4 (2010), pages 271–302. DOI: 10.1017/S095679681000016X.

[9] Garrett Birkhoff. "On the Structure of Abstract Algebras". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 31.4 (1935), pages 433–454. DOI: 10.1017/S0305004100013463.

[10] Garrett Birkhoff and John D. Lipson. "Heterogeneous Algebras". In: *Journal of Combinatorial Theory* 8.1 (1970), pages 115–133. DOI: 10.1016/S0021-9800(70)80014-X.

[11] Corrado Böhm and Alessandro Berarducci. "Automatic synthesis of typed Λ-programs on term algebras". In: *Theoretical Computer Science* 39 (1985), pages 135–154. DOI: 10.1016/0304-3975(85)90135-5.

[12] Val Breazu-Tannen and Jean Gallier. "Polymorphic Rewriting Conserves Algebraic Confluence". In: *Information and Computation* 114.1 (1994), pages 1–29. DOI: 10.1006/inco.1994.1078.

[13] Val Breazu-Tannen and Jean Gallier. "Polymorphic Rewriting Conserves Algebraic Strong Normalization". In: *Theoretical Computer Science* 83.1 (1991), pages 3–28. DOI: 10.1016/0304-3975(91)90037-3.

[14] Janusz A. Brzozowski. "Derivatives of Regular Expressions". In: *Journal of the ACM* 11.4 (1964), pages 481–494. DOI: 10.1145/321239.321249.

[15] Stanley Burris and H. P. Sankappanavar. *A Course in Universal Algebra*. Volume 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.

[16] Rod M. Burstall. "Proving Properties of Programs by Structural Induction". In: *The Computer Journal* 12.1 (1969), pages 41–48. DOI: 10.1093/comjnl/12.1.41.

[17] Luca Cardelli. "Type Systems". In: *The Computer Science and Engineering Handbook*. Edited by Allen B. Tucker. CRC Press, 1997, pages 2208–2236.

[18] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. "Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages". In: *Journal of Functional Programming* 19.5 (2009), pages 509–543. DOI: 10.1017/S0956796809007205.

[19] 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.

[20] Wyatt Childers, Peter Dimov, Barry Revzin, and Daveed Vandevoorde. *define_static_{string,object,array}*. Technical report P3491R3. ISO/IEC JTC1/SC22/WG21, 2025. URL: https://wg21.link/P3491R3.

[21] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Communications of the ACM* 13.6 (1970), pages 377–387. DOI: 10.1145/362384.362685.

[22] Charles Consel and Olivier Danvy. “Partial Evaluation of Pattern Matching in Strings”. In: *Information Processing Letters* 30.2 (1989), pages 79–86. DOI: 10.1016/0020-0190(89)90113-0.

[23] 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, pages 238–252. DOI: 10.1145/512950.512973.

[24] Ivan Čukić. *Functional Programming in C++*. Manning Publications, 2018. ISBN: 9781617293818.

[25] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Transactions on Programming Languages and Systems* 13.4 (1991), pages 451–490. DOI: 10.1145/115372.115320.

[26] Luis Damas and Robin Milner. “Principal Type-Schemes for Functional Programs”. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1982, pages 207–212. DOI: 10.1145/582153.582176.

[27] Richard Dedekind. *Was sind und was sollen die Zahlen?* Braunschweig: Vieweg, 1888.

[28] Edsger W. Dijkstra. *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1976.

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

[30] Gabriel Dos Reis and Bjarne Stroustrup. “Specifying C++ concepts”. In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2006, pages 295–308. DOI: 10.1145/1111037.1111064.

[31] Stephen Ellis, Shuofei Zhu, Nobuko Yoshida, and Linhai Song. “Generic Go to Go: Dictionary-passing, monomorphisation, and hybrid”. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (2022), 168:1–168:29. DOI: 10.1145/3563331.

[32] Matthias Felleisen. “On the Expressive Power of Programming Languages”. In: *Science of Computer Programming* 17.1–3 (1991), pages 35–75. DOI: 10.1016/0167-6423(91)90036-W.

[33] Matteo Frigo and Steven G. Johnson. “The Design and Implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (2005), pages 216–231. DOI: 10.1109/JPROC.2004.840301.

[34] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. “An extended comparative study of language support for generic programming”. In: *Journal of Functional Programming* 17.2 (2007), pages 145–205. DOI: 10.1017/S0956796806006198.

[35] *GHC User’s Guide: Overlapping Instances*. The GHC Team. 2024. URL: https://downloads.haskell.org/ghc/9.10.1/docs/users_guide/exts/instances.html#overlapping-instances.

[36] Jeremy Gibbons. “The Third Homomorphism Theorem”. In: *Journal of Functional Programming* 6.4 (1996), pages 657–665. DOI: 10.1017/S0956796800001908.

[37] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Volume 7. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[38] Joseph A. Goguen and José Meseguer. “Completeness of Many-Sorted Equational Logic”. In: *Houston Journal of Mathematics* 11.3 (1985), pages 307–334.

[39] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. “Concepts: linguistic support for generic programming in C++”. In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2006, pages 291–310. DOI: 10.1145/1167473.1167499.

[40] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. “Featherweight Go”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), 149:1–149:29. DOI: 10.1145/3428217.

[41] Thomas Hubert, Rishi Mehta, Laurent Sartran, et al. “Olympiad-level formal mathematical reasoning with reinforcement learning”. In: *Nature* 651.8106 (2026), pages 607–613. DOI: 10.1038/s41586-025-09833-y.

[42] Paul Hudak. “Building Domain-Specific Embedded Languages”. In: *ACM Computing Surveys* 28.4es (1996), page 196. DOI: 10.1145/242224.242477.

[43] Gérard Huet. “The Zipper”. In: *Journal of Functional Programming* 7.5 (1997), pages 549–554. DOI: 10.1017/S0956796897002864.

[44] Graham Hutton. “A Tutorial on the Universality and Expressiveness of Fold”. In: *Journal of Functional Programming* 9.4 (1999), pages 355–372. DOI: 10.1017/S0956796899003500.

[45] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. “Featherweight Java: a minimal core calculus for Java and GJ”. In: *ACM Transactions on Programming Languages and Systems* 23.3 (2001), pages 396–450. DOI: 10.1145/503502.503505.

[46] *ISO/IEC 9075-2: Information Technology, Database Languages SQL, Part 2: Foundation (SQL/Foundation)*. Technical report ISO/IEC 9075-2:2023. International Organization for Standardization, 2023. URL: https://www.iso.org/standard/76584.html.

[47] 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.

[48] Mark P. Jones. *Qualified Types: Theory and Practice*. Volume 9. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994. DOI: 10.1017/CBO9780511663086.

[49] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993. ISBN: 0-13-020249-5.

[50] Paul Keir and Hana Dusíková. *constexpr std::shared_ptr and friends*. Technical report P3037R6. Adopted into the C++26 working draft, June 2025. ISO/IEC JTC1/SC22/WG21, May 2025. URL: https://open-std.org/JTC1/sc22/wg21/docs/papers/2025/p3037r6.pdf.

[51] Richard A. Kelsey. "A correspondence between continuation passing style and static single assignment form". In: *ACM SIGPLAN Notices* 30.3 (1995). ACM SIGPLAN Workshop on Intermediate Representations, pages 13–22. DOI: 10.1145/202530.202532.

[52] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. 2nd edition. Section 10.1, Generic Data Types; the text is also published at https://doc.rust-lang.org/book/. San Francisco: No Starch Press, 2023.

[53] Donald E. Knuth. "Semantics of Context-Free Languages". In: *Mathematical Systems Theory* 2.2 (1968), pages 127–145. DOI: 10.1007/BF01692511.

[54] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3rd edition. Reading, MA: Addison-Wesley, 1997.

[55] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd edition. Reading, MA: Addison-Wesley, 1997.

[56] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. "Fast Pattern Matching in Strings". In: *SIAM Journal on Computing* 6.2 (1977), pages 323–350. DOI: 10.1137/0206024.

[57] 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 2026-10-24).

[58] Matthias Kretz. *std::simd: Merge Data-Parallel Types from the Parallelism TS 2*. Technical report P1928R15. ISO/IEC JTC1/SC22/WG21, 2024. URL: https://wg21.link/P1928R15.

[59] Jeffrey C. Lagarias. "The $3x+1$ Problem and its Generalizations". In: *The American Mathematical Monthly* 92.1 (1985), pages 3–23. DOI: 10.1080/00029890.1985.11971528.

[60] Joachim Lambek. "A fixpoint theorem for complete categories". In: *Mathematische Zeitschrift* 103.2 (1968), pages 151–161. DOI: 10.1007/BF01110627.

[61] Peter J. Landin. “The Mechanical Evaluation of Expressions”. In: *The Computer Journal* 6.4 (1964), pages 308–320. DOI: 10.1093/comjnl/6.4.308.

[62] John Launchbury and Simon L. Peyton Jones. “Lazy Functional State Threads”. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Orlando, Florida: ACM, 1994, pages 24–35. DOI: 10.1145/178243.178246.

[63] Gottfried Wilhelm Leibniz. “Discours de métaphysique”. In: *Sämtliche Schriften und Briefe, Reihe VI: Philosophische Schriften, Band 4: 1677–Juni 1690*. Written 1686, N. 306; the identity of indiscernibles is §9. Berlin: Akademie Verlag, 1999, pages 1529–1588.

[64] Gottfried Wilhelm Leibniz. “Discourse on Metaphysics”. In: *Philosophical Essays*. Edited by Roger Ariew and Daniel Garber. Written 1686; the identity of indiscernibles is §9. Indianapolis: Hackett, 1989, pages 35–68.

[65] Jack Lindsey, Wes Gurnee, Emmanuel Ameisen, Brian Chen, Adam Pearce, Nicholas L. Turner, Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar, Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan, Adam Jermyn, Andy Jones, Andrew Persic, Zhenyi Qi, T. Ben Thompson, Sam Zimmerman, Kelley Rivoire, Thomas Conerly, Chris Olah, and Joshua Batson. *On the Biology of a Large Language Model*. Transformer Circuits Thread. Mar. 2025. URL: https://transformer-circuits.pub/2025/attribution-graphs/biology.html.

[66] Simon Marlow, editor. *Haskell 2010 Language Report*. 2010. URL: https://www.haskell.org/onlinereport/haskell2010/.

[67] Ralph Matthes. “Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types”. PhD thesis. Ludwig-Maximilians-Universität München, 1999. URL: https://www.irit.fr/~Ralph.Matthes/dissertation/matthes_diss.pdf.

[68] Barry Mazur. “When is one thing equal to some other thing?” In: *Proof and Other Dilemmas: Mathematics and Philosophy*. Edited by Bonnie Gold and Roger A. Simons. Spectrum. Mathematical Association of America, 2008, pages 221–242.

[69] Erik Meijer, Maarten Fokkinga, and Ross Paterson. “Functional programming with bananas, lenses, envelopes and barbed wire”. In: *Conference on Functional Programming Languages and Computer Architecture*. Volume 523. Lecture Notes in Computer Science. Springer, 1991, pages 124–144. DOI: 10.1007/3540543961_7.

[70] Bartosz Milewski. *Category Theory for Programmers*. Compiled from the original blog series at https://bartoszmilewski.com. Blurb, Incorporated, 2019. ISBN: 9780464243878.

[71] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and System Sciences* 17.3 (1978), pages 348–375. DOI: 10.1016/0022-0000(78)90014-4.

[72] *MISRA C++:2023: Guidelines for the Use of C++17 in Critical Systems*. The MISRA Consortium Limited. 2023.

[73] J. Garrett Morris and Mark P. Jones. "Instance Chains: Type Class Programming Without Overlapping Instances". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2010, pages 375–386. DOI: 10.1145/1863543.1863596.

[74] Leonardo de Moura. *Postmortem for Kernel Soundness Bug #14576*. 2026. URL: https://leodemoura.github.io/blog/2026-8-1-postmortem-for-kernel-soundness-bug-14576/ (visited on 2026-11-05).

[75] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 theorem prover and programming language". In: *Automated Deduction – CADE 28*. Lecture Notes in Computer Science. Springer, 2021, pages 625–635. DOI: 10.1007/978-3-030-79876-5_37.

[76] Jonathan Müller. *A Principled Approach to Safety Profiles*. Technical report P3649R0. ISO/IEC JTC1/SC22/WG21, 2025. URL: https://open-std.org/JTC1/SC22/WG21/docs/papers/2025/p3649r0.html.

[77] nLab authors. *retract*. ncatlab.org/nlab/show/retract, accessed 2026-10-27. URL: https://ncatlab.org/nlab/show/retract.

[78] Giuseppe Peano. *Arithmetices principia, nova methodo exposita*. Turin: Bocca, 1889.

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

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

[81] Benjamin C. Pierce and David N. Turner. "Local Type Inference". In: *ACM Transactions on Programming Languages and Systems* 22.1 (2000), pages 1–44. DOI: 10.1145/345099.345100.

[82] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. "Verified Low-Level Programming Embedded in F*". In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017), 17:1–17:29. DOI: 10.1145/3110261.

[83] Markus Püschel, José M. F. Moura, Jeremy R. Johnson, et al. "SPIRAL: Code Generation for DSP Transforms". In: *Proceedings of the IEEE* 93.2 (2005), pages 232–275. DOI: 10.1109/JPROC.2004.840306.

[84] Dimi Racordon, Eugene Flesselle, and Cao Nguyen Pham. "On the State of Coherence in the Land of Type Classes". In: *The Art, Science, and Engineering of Programming* 10.1 (2025), 15:1–15:25. DOI: 10.22152/programming-journal.org/2025/10/15.

[85] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2013, pages 519–530. DOI: 10.1145/2491956.2462176.

[86] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. "A mechanized semantics for C++ object construction and destruction, with applications to resource management". In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2012, pages 521–532. DOI: 10.1145/2103656.2103718.

[87] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. "Formal verification of object layout for C++ multiple inheritance". In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2011, pages 67–80. DOI: 10.1145/1926385.1926395.

[88] John C. Reynolds. "Types, abstraction and parametric polymorphism". In: *Information Processing 83*. Edited by R. E. A. Mason. North-Holland, 1983, pages 513–523.

[89] Cynthia Rudin. "Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead". In: *Nature Machine Intelligence* 1.5 (2019), pages 206–215. DOI: 10.1038/s42256-019-0048-x.

[90] Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. EATCS Monographs in Theoretical Computer Science. Springer, 2012. DOI: 10.1007/978-3-642-17336-3.

[91] William Shakespeare. *An Excellent Conceited Tragedie of Romeo and Iuliet*. Act 2, Scene 2. John Danter, 1597.

[92] Jeremy Siek and Andrew Lumsdaine. "Essential language support for generic programming". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2005, pages 73–84. DOI: 10.1145/1065010.1065021.

[93] Jeremy Siek and Andrew Lumsdaine. *Essential language support for generic programming: Formalization part 1*. Technical report TR605. Indiana University, 2004.

[94] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley, 2009.

[95] Christopher Strachey. "Fundamental Concepts in Programming Languages". In: *Higher-Order and Symbolic Computation* 13.1–2 (2000). Lecture notes, Copenhagen, 1967, pages 11–49. DOI: 10.1023/A:1010000313106.

[96] Bjarne Stroustrup. "Abstraction and the C++ Machine Model". In: *Embedded Software and Systems (ICESS 2004)*. Volume 3605. Lecture Notes in Computer Science. Springer, 2005, pages 1–13. DOI: 10.1007/11535409_1.

[97] Bjarne Stroustrup. *Creator of C++: Bell Labs, Negative Overhead Abstraction, Mistakes*. The Peterman Pod, interview by Ryan Peterman, 18 May 2026, at 1:23:12. May 2026. URL: https://www.developing.dev/p/creator-of-c-bell-labs-negative-overhead.

[98] Bjarne Stroustrup. *The C++ox "Remove Concepts" Decision*. Dr. Dobb's Journal. July 2009.

[99] Bjarne Stroustrup. "Thriving in a Crowded and Changing World: C++ 2006–2020". In: *Proceedings of the ACM on Programming Languages* 4.HOPL (2020), 70:1–70:167. DOI: 10.1145/3386320.

[100] Bjarne Stroustrup and Gabriel Dos Reis. *Concepts – Design Choices for Template Argument Checking*. Technical report N1522 = 03-0105. ISO/IEC JTC1/SC22/WG21, Oct. 2003. URL: https://www.stroustrup.com/N1522-concept-criteria.pdf.

[101] Bjarne Stroustrup and Herb Sutter, editors. *C++ Core Guidelines*. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines. 2026.

[102] Martin Sulzmann, Gregory J. Duck, Simon Peyton Jones, and Peter J. Stuckey. "Understanding functional dependencies via constraint handling rules". In: *Journal of Functional Programming* 17.1 (2007), pages 83–129. DOI: 10.1017/S0956796806006137.

[103] Herb Sutter. *C++26 is done! Trip report: March 2026 ISO C++ standards meeting (London Croydon, UK)*. 2026. URL: https://herbsutter.com/2026/03/29/c26-is-done-trip-report-march-2026-iso-c-standards-meeting-london-croydon-uk/ (visited on 2026-11-06).

[104] Herb Sutter. *Core Safety Profiles for C++26*. Technical report P3081R2. ISO/IEC JTC1/SC22/WG21, 2025. URL: https://isocpp.org/files/papers/P3081R2.pdf.

[105] Terence Tao. *Palomar – a registry of Lean verified mathematics*. What's new (blog), 18 August 2026, and the author's replies in its discussion. Aug. 2026. URL: https://terrytao.wordpress.com/2026/08/18/palomar-a-registry-of-lean-verified-mathematics/.

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

[107] The LLVM Project. *libc++ C++23 Status*. P2322R6, ranges fold algorithms: complete in release 23. URL: https://libcxx.llvm.org/Status/Cxx23.html (visited on 2026-10-24).

[108] Ken Thompson. "Reflections on Trusting Trust". In: *Communications of the ACM* 27.8 (1984), pages 761–763. DOI: 10.1145/358198.358210.

[109] Alan M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pages 230–265. DOI: 10.1112/plms/s2-42.1.230.

[110] David A. Turner. "Total Functional Programming". In: *Journal of Universal Computer Science* 10.7 (2004), pages 751–768. DOI: 10.3217/jucs-010-07-0751.

[111] Miles Turpin, Julian Michael, Ethan Perez, and Samuel R. Bowman. "Language Models Don't Always Say What They Think: Unfaithful Explanations in Chain-of-Thought Prompting". In: *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*. 2023. arXiv: 2305.04388.

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

[113] Philip Wadler and Stephen Blott. "How to make ad-hoc polymorphism less ad hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM. 1989, pages 60–76. DOI: 10.1145/75277.75283.

[114] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. "Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs". In: *16th USENIX Symposium on Networked Systems Design and Implementation*. 2019. URL: https://www.usenix.org/conference/nsdi19/presentation/wang-xiang.

[115] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. "An operational semantics and type safety proof for multiple inheritance in C++". In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2006, pages 345–362. DOI: 10.1145/1167473.1167503.

About the author

Vincent Kräutler holds a PhD in computational biophysics from ETH Zürich, Switzerland, and enjoys working on software that has to be fast. Contact him at vincent.kraeutler@alumni.ethz.ch.
https://orcid.org/0009-0006-7317-3502