Programming Languages as Intermediate Representations in the AI-Generated-Code Era

Constantin Rack *

2026

Abstract

Code is becoming what assembly became forty years ago: an intermediate representation. The author of new code is increasingly not a human, and in the design center of the next decade, the author of new code is also not an AI that writes code in any meaningful sense; code is being synthesized from intent. Humans express intent; a system produces code; the code is read by neither party in the sense in which it once was. Programming language design has, throughout its history, optimized for one constraint: a human reads and writes the code, and every prevailing language feature is downstream of it. We argue that this is not an incremental shift to be accommodated by retrofitting tooling, but a categorical change in the design center of programming languages: when code is an intermediate representation, the constraints that produced today's language landscape no longer bind. We claim that three properties should occupy the resulting design space: a deliberately small surface, a fully mechanized static semantics, and a compilation chain (from intent through code to machine) that is end-to-end a formal object.

1 Introduction

For most of the history of programming, code has been the artifact humans write and read, the medium through which intent is expressed, captured, audited, and modified. We argue that this role is dissolving.

The premise of this paper, stated as five propositions:

P1. No human routinely reads code anymore.

P2. No AI writes code in any meaningful sense.

P3. Code is the new assembly: an intermediate representation.

P4. Humans express intent.

P5. Intent synthesizes code.

*Independent Researcher. constantin@rack.li · www.constantin-rack.com

P1, P3, P4, and P5 are not introduced here as novel framings. P1, that humans no longer routinely read the code in their stack, is empirically documented in studies of "vibe-coded" development practices (*Vibe Coding Needs Vibe Reasoning* 2025; Töpfer et al. 2026), in surveys of developer behaviour (Gurgul et al. 2025), and in security audits of LLM-generated code (Soltanian Fard Jahromi et al. 2026); the vibe-reasoning study reports 1.7× more major issues, 75% more logic errors, and 2.74× more security vulnerabilities in unreviewed AI code than in human-authored code at comparable scale. P3, that code is best understood as an intermediate representation rather than a user interface (the role assembly came to occupy in compilation pipelines once higher-level source languages displaced it), has been argued explicitly in recent industrial and academic work on agentic-era software conventions (Trooskens et al. 2026; Ustynov 2026). P4 and P5, that humans express intent and intent synthesizes code, describe an architecture that AI-friendly languages (Fei et al. 2024; Mell et al. 2025) and intent-to-spec pipelines (Ye et al. 2026; Feng et al. 2026) already operationalize, and which recent software-engineering-community discourse traces as a maturity pathway from individual agent assistance to bounded-autonomous engineering (De La Cruz 2026). The pattern has industrial precedents predating LLM-based code generation: the iMatix Generator Scripting Language (iMatix Corporation 1995) demonstrated model-to-code synthesis at production scale without formal correctness guarantees; T2 and T3 address precisely the gap that achievement left open. P2, the proposition most contestable against current common sense, is the one we defend most carefully.

Three terms in this paper carry a specific meaning.

Design center. The constellation of users and constraints around which a programming language is optimized. The recentering thesis is that this constellation has shifted from the human author of code to the synthesizer-and-verifier pair on either side of an internal intermediate representation.

Intent. A specification of what a program should do, expressed at a level above code. The form of intent (natural-language description, formal specification, mixed declarative and example-based, or other) is not fixed by our argument, only its location at the top of the chain.

Synthesizer. Any system that translates intent to code. We treat it as a single conceptual stage regardless of whether the underlying mechanism is a learned model, a rule-based translation from a structured intent specification, or a combination of the two.

P2 will read as paradoxical against the current common sense. Large language models (LLMs) demonstrably produce code today, in volume, and the empirical record of that production is well documented: systematic reviews catalog quality issues across nine independent dimensions in 114 primary studies (He et al. 2026), and developer surveys report that the locus of expert work is migrating from writing code to writing specifications and architectural decisions (Gurgul et al. 2025). The claim of P2 is structural, not empirical. "AI writes code" is the framing of a transitional stage in which the LLM substitutes for a human author and is judged by how closely its output resembles human-authored code. In the design center we describe, that framing breaks: there is no human author to substitute for, and the LLM operates as a translation step between intent and code in the way a compiler operates as a translation step between source and target. Compilers do not "write" machine code; they synthesize it from a higher-level representation. The same will hold of the synthesizer at the top of the chain. The transitional stage is real and ongoing; the design center we describe is the stage that follows.

The argument that the bottleneck in software is *essential* difficulty (specification, conformity, changeability) rather than *accidental* difficulty (notation, tools, syntax) is a framework commonly invoked against productivity claims for code automation (Brooks 1987). Our argument is consistent with that framework, not opposed

to it: code generation attacks accidental difficulty, where Brooks himself acknowledged that gains had been made; the essential work (intent specification) remains with the human, where Brooks said no silver bullet exists.

This recentering is not yet universally true. It is true at the margin and at scale, and the margin is where the language design questions of the next decade are being decided. We do not litigate the premise here. We take it as the starting condition for the question that follows: *what does a programming language look like when its surface is no longer the boundary between human and machine, but an internal representation between two levels neither of which is human?*

Every prevailing feature of mainstream programming languages was a response to a human reader and a human writer. Short keywords exist because human typing is slow. Type inference exists primarily because declaring types is tedious for humans. Syntactic sugar exists because the desugared form is too long for humans to write comfortably. Overloading and implicit conversion exist because humans lose track of distinct names for related operations. Ergonomic error messages exist because humans need their failures explained in their language. Macros exist because humans tire of repetition. The list is not incidental. It is the list of every major mainstream language feature, and it is, almost without exception, an ergonomic accommodation of the human author of code.

When code becomes an intermediate representation, the accommodation no longer applies. A language designed to be read and written by humans is, in the new architecture, a language designed for a constituency that has moved up one level. The human writes intent above; the synthesizing system operates on whatever surface is most convenient for it; the executing machine cares only about the lowered output. The design choices that were once forced (readability, conciseness, ergonomic surface) are not merely optional. They are optimizing for a constituency that no longer exists at the surface in question. The freed design space can and must be put to use elsewhere.

We claim three things, and we defend them in §3, §4, and §5 respectively.

T1 (Surface). *Surface ergonomics is a cost, not a benefit, when code is an intermediate representation between intent and machine.* A small, deliberately explicit surface is preferable to a large, ergonomic one. The cost-benefit analysis that justified ergonomic surfaces no longer holds, because the constituency it served has moved one level up.

T2 (Mechanized Metatheory). *Fully mechanized type-system metatheory is no longer aspirational; it is the new floor.* The arguments that historically positioned mechanized metatheory as a research luxury were arguments about cost. The cost has shifted; the position should shift with it. An intermediate representation (IR) that no human reads is, by structural analogy with every other IR in computing, the natural level at which to demand full formal verification.

T3 (Pipeline). *The compilation chain, from intent through code to machine, must be a formal object end to end, not a sequence of compilers.* When humans write intent and machines execute the lowered output, every translation in between must carry its correctness as a formal artifact. Compilation, treated since its origins as an engineering matter to be addressed by testing, must become a property the chain carries with itself, at every stage.

The three claims are not independent: T1 grants the small surface that makes T2 tractable, and the verified surface that T2 produces is the foundation T3 builds the chain on. We argue them in sequence because that is the order of dependency; we defend them as a system because that is the form in which the design center

holds.

We note one piece of inheritance work the field has already done. Verified compilation has been demonstrated as research, most prominently by CompCert (Leroy 2009) for C and CakeML (Kumar et al. 2014) for ML. Industrial-style precedents for the broader architecture we describe, with LLM-assisted generation at a compilation stage and deterministic execution thereafter, have also begun to appear (Trooskens et al. 2026; Mell et al. 2025). A production instantiation of the T1+T2+T3 stack, AWS's Cedar policy language with Lean 4 formalization and differential testing of the reference implementation (Cedar Policy team 2026), demonstrates that the overhead of full formalization is acceptable when the language surface is deliberately constrained. The verified-systems tradition has now reached the deployed runtime as well: seL4 (Klein et al. 2009) established more than a decade ago that an entire operating-system kernel can be machine-checked against a formal specification, completing the picture at the layer below verified compilation. The transitive structure of trust this composition requires was first articulated by Thompson (1984), whose argument that partial trust in a toolchain collapses to no trust remains the canonical statement of why T3's end-to-end requirement is not optional. Recent technical work on phased semantic preservation (Haynes 2026) and Curry-Howard correspondences for resource cleanup (Congard et al. 2025) supplies further pieces of the same picture: each translation stage carries explicit invariants forward, and each invariant is mechanically discharged. T3 is not, in its individual claim, novel. The novelty, and what this paper argues for, is the combination of T1, T2, and T3 as the normal case for a programming language, made tractable precisely because the human-author constraint on code itself has been relaxed and the resulting language is no longer required to be ergonomic for any party. This aligns with a thesis recently articulated, from a different vantage in a 2026 essay, by Moura (2026), who asks who verifies the world's software when AI writes it.

2 The Constraint That No Longer Binds

Programming language design has, over six decades, accumulated a set of features so thoroughly normalized that they are no longer recognized as design choices. They are recognized as language. Below we re-examine them, one by one, as design choices, each of which had a reason, each of which served a constraint, and each of which can now be reconsidered because the constraint they served is no longer in force.

Brevity. Every major language has short keywords: `if`, `for`, `let`, `def`, `fn`. The reason is not aesthetic. Human typing is slow; human reading skims faster across short tokens than long ones; human screen real estate is finite. None of these constraints applies to a machine. A machine emits long tokens as readily as short ones. More: a machine pattern-matching against a long, distinctive token does so more reliably than against a short, ambiguous one. Brevity, for a machine author, is not a benefit. It is a source of confusion.

Inference. Type inference is justified almost entirely by the cost it saves the human writer. An expression's type is, in principle, deducible from context; demanding that the human declare it anyway is an annoyance, and inference relieves it. For a machine, the cost is not symmetric. A machine writing inferred types and a machine writing explicit types are doing roughly equivalent work. But the *costs of inference* are not symmetric either. Inference adds error-message ambiguity (the compiler reports a type mismatch involving a type the user never wrote), it adds compilation-time overhead, and it makes mechanical reasoning about programs harder. The benefit-side of the ledger has shrunk; the cost-side has not.

Syntactic sugar. Sugar is the explicit expression of the human-author bias. A construct is sugared because the underlying form is "too verbose," meaning too verbose *for a human to write*. A machine writes the underlying form as readily as the sugared form. More: every desugaring rule is a place where the meaning of

source code does not coincide with its surface, and a place where a downstream tool (a verifier, a debugger, a linter, a code-review tool) must either replicate the desugaring or accept that it is reasoning about source that does not quite match what runs. Sugar is, for the human, ergonomics. For the machine, it is noise around the actual semantics.

Overloading and implicit conversion. Overloading exists because humans experience symbol scarcity: finding many distinct, mnemonically reasonable names for related operations is hard, and the human reader is helped by the consistency of using the same symbol. Implicit conversion exists because humans forget which type they have. Both place an interpretive burden on every reader of the resulting code, including any tool that processes it. A machine does not experience symbol scarcity; it does not forget types; and a tool that has to disambiguate an overloaded operator on the way to verifying a property is doing extra work to recover information the human author chose not to write. The trade was favorable when it bought the human's comfort. It is unfavorable when it buys nothing and costs everything downstream.

Ergonomic error messages. Error messages are the most visible point at which a compiler addresses its user. A modern compiler error message is a careful piece of human-language text designed to tell a human reader what went wrong, why, and how to fix it. The machine that consumes the message takes none of this. It takes structured data: an error code, a location, perhaps a suggested edit, ideally a representation of the failed proof obligation or the unification conflict. The work that goes into shaping a human-readable error is, in the new world, work that produces nothing for the consumer.

Macros and metaprogramming. Macros exist so the human can write less code. The use cases (repetitive boilerplate, abstraction over patterns the type system cannot express, embedded domain-specific languages) are all responses to the cost the human pays for repetition or abstraction. A machine does not pay that cost. A machine generating ten similar functions does not feel the boredom that drove the human to write a macro that generates them. And macros, like sugar, place a layer between the source and its meaning that every downstream tool must either traverse or accept ignorance of.

The cumulative observation is direct. The standard menu of mainstream language features is, with negligible exception, an ergonomic accommodation of the human author of code. When code becomes an intermediate representation, the human author of code disappears entirely: the human writes intent at one level above; the machine executes lowered output at one level below; the menu of features at the code surface becomes a list of accommodations with no constituency. The accommodations they provided are no longer needed. The costs they imposed (on tooling, on reasoning, on verification, on the size and complexity of every implementation) remain. The trade has reversed.

3 The First Claim: Surface Ergonomics is a Cost (T1)

We claim that, in a world where code is an intermediate representation between intent and machine, a deliberately small surface, measured by both the count of reserved constructs and the absence of derived forms, is preferable to a large, ergonomic one. The claim is not that small surfaces have small advantages; it is that the cost-benefit analysis that justified large surfaces has reversed.

The argument that smaller languages are more reliable than larger ones is old (Hoare 1973). A complementary argument from the same era (Dijkstra 1978) warns against the opposite impulse, the desire to make programming languages more like natural language: formal precision is exactly the property programming notation supplies and that natural language lacks; the case for retaining and sharpening that precision strengthens, not weakens, when the writer is a machine. The minimalism tradition continues into the late twentieth

century (Wirth 1995), where the discipline of refusal (leaving features out) is named as the most undervalued language-design skill, with Modula-2 and Oberon offered as evidence that minimal designs are not less capable but differently capable. The change is that the historical pressure against minimalism, the cost it imposed on human authors, has dissolved, and the case for small surfaces no longer needs to be made against a competing case. Recent work on language design specifically for AI generation (Fei et al. 2024) reports that mandatory type signatures and reduced structural complexity, choices that would be ergonomically taxing for human authors, measurably improve generation quality, providing early empirical support for the case we make here on theoretical grounds.

There are two reasons. The first concerns the machine that writes the code. The second concerns the machine that verifies it. The two reasons reinforce each other. Three objections, addressed in turn below, follow.

Why small surface helps the machine that writes. A machine generating code from a higher-level intent is, in effect, performing a translation between distributions: from a distribution of intents to a distribution of programs. Translation accuracy depends on the regularity of the target distribution. A language with many ways to express the same thing (sugar, overloading, alternative idioms) has high target-distribution variance, and a machine learning to map onto it is learning a more difficult function. A language with one way to express each thing has low target-distribution variance, and the machine's function is correspondingly simpler. Robustness improves; ambiguity diminishes; the failure mode of the generator (producing valid code with wrong semantics) becomes rarer because there are fewer valid forms whose semantics could be confused. None of this argument depends on the specific generator architecture. It depends only on the observation that ambiguity in the target hurts translation accuracy, and ambiguity in the target is precisely what surface ergonomics introduces. Empirical support comes from work on language design specifically for AI generation (Fei et al. 2024), which reports that constraints reducing target-distribution variance measurably improve generation quality.

Why small surface helps the machine that verifies. Mechanized metatheory (the formal verification, in a proof assistant, of properties of a language's type system and operational semantics) has a cost that scales with surface area. Every additional construct adds typing rules, reduction rules, and metatheoretic obligations. Every additional desugaring adds a soundness obligation: that the desugared form preserves the source's semantics. Every overloading rule adds a disambiguation obligation. The cost is super-linear in the number of features, because new features interact with old ones, and each interaction is itself a proof obligation. Halving the surface area is therefore not a marginal improvement to the verification effort. It is a structural one. A surface small enough to mechanize fully is different in kind from a surface that is "almost small enough": the former admits a complete metatheory; the latter admits only partial guarantees, and partial guarantees in a verification setting are nearly as bad as none, because they leave open exactly the cases an adversarial input is likely to exploit. The cost-with-surface relation is itself a documented engineering reality: the POPLmark Challenge (Aydemir et al. 2005) established that mechanizing the metatheory of even a deliberately small calculus is a substantial effort, and the intervening two decades of follow-up work have not closed the gap for languages of mainstream-grammar size.

The first objection: humans still read the code. We do not deny it. Humans read code today, and humans will continue to read code in the new design center, for audit, for incident response, for the rare cases where intent-level review is insufficient. We claim that this readership is becoming what assembly readership became: rare, specialized, and exercised by humans who are at that moment auditing rather than authoring. For an auditing reader, *explicit* forms are easier than *terse* forms: they cannot rely on muscle memory or pattern recognition, and they read each construct as new. The surface that optimizes for the assembly-level auditor

of today (clear, regular, no clever encodings) is the surface that should optimize for the code-level auditor of tomorrow. Ergonomic surfaces are optimized for the *frequent* reader; reframing the human as occasional auditor rather than primary author changes the optimization target. Recent work that formalizes auditability as a distinct design dimension (Nian et al. 2026) supports this reframing: action recoverability, lifecycle coverage, policy checkability, responsibility attribution, and evidence integrity are properties of the audit interface, not of the authoring interface, and they impose different design constraints.

The second objection: this trades human cost for machine cost. It does. The trade is in the right direction. Machine cost amortizes across all programs the system generates; the cost of a longer source representation is paid once, by the generator, in compute that is approximately free at the margin. Human cost was the bottleneck of programming language history; relieving it cost the language designer concessions across the board. Spending machine cost to recover those concessions is a favorable trade; the same economic logic, formalized through Transaction Cost Economics, underwrites the case for upfront specification governance over after-the-fact remediation (Farrag 2026). We acknowledge that the trade has limits (we are not arguing that languages should become arbitrarily verbose for arbitrary marginal gains in machine clarity), but the regime in which we currently operate is far from those limits, and the marginal step toward smaller surface remains favorable.

The third objection: a small surface restricts what can be expressed. This is an empirical claim, and the historical record does not support it. Every feature mainstream languages have evolved to express was, at some point, expressible in a more primitive language without that feature; the encoding-into-core-calculi tradition that established this in the type-theoretic case is summarized at textbook length in (Pierce 2002). The reasons for the evolution were ergonomic: the primitive form was tedious, error-prone, or unidiomatic for a human. None of these reasons holds for a machine. Mainstream language features have, with few exceptions, added expressive convenience for human authors rather than expressive power that was not already encodable in the language's underlying calculus. T1 is the claim that, in the new design center, expressive convenience for human authors is no longer something to optimize for, and the features that exist solely to provide it are precisely the features that should be removed.

The argument for T1 closes with the observation that surface size is the load-bearing variable for the rest of the paper. T2 (that mechanized metatheory is the new floor) is feasible only because T1 is granted; mechanizing the metatheory of a fully ergonomic mainstream language is infeasible at any reasonable cost, and has been the entire time. T3 (that the compilation pipeline must be a formal object) is similarly downstream: a pipeline whose source language has minimal surface admits formal treatment of each translation step in a way that a pipeline with a sugared, overloaded, inferring source language never has. T1 is the prerequisite. We grant ourselves the prerequisite by recognizing that the constraint that historically forbade it has been relaxed.

4 The Second Claim: Mechanized Metatheory is the New Floor (T2)

We claim that, given T1, the level of confidence a programming language ought to provide in its own static semantics has shifted upward by a category. What was previously an expensive research artifact, a complete mechanization of a language's type system and operational semantics with machine-checked proofs of soundness, is now achievable as ordinary engineering, and accordingly should be expected as ordinary engineering. The work remains costly, but the cost is no longer prohibitive enough to make it optional.

By "mechanized metatheory" we mean the development, in a proof assistant, of a formal model of a programming language: the syntax of its expressions, the rules of its type system, the rules of its operational semantics,

and the principal soundness theorems (type preservation, progress, and any language-specific safety guarantees the language wishes to claim) (Pierce 2002). The intellectual lineage is older: the operational tradition of giving programs precise mathematical meanings began with Floyd (1967), and the dependent-type-theoretic framework in which contemporary proof assistants formulate their soundness arguments traces to Martin-Löf (1984). The mechanization is *machine-checked*: every step of every proof is verified by software whose own correctness is the subject of a separate, much smaller, body of work (Carneiro 2024). The soundness theorems, once mechanized, do not depend on the careful attention of any human auditor for their reliability. They depend on the correctness of the proof assistant's kernel, and on the faithfulness of the formal model to the language as actually compiled and executed. Both dependencies are addressable, and have been addressed in the literature.

Mechanized metatheory has been pursued as research for thirty years. It has not been the default for production languages, for three reasons. The first is that mainstream language surfaces are too large to mechanize completely; partial mechanizations are possible, but partial mechanizations leave open precisely the corners that are most likely to harbor unsoundness, which is to say the corners that are least useful to leave open. The second is that the features mainstream languages add for human-author ergonomics (sugar, overloading, implicit conversion, complex type inference) are disproportionately hard to model formally; their semantic content is distributed and contextual rather than local and explicit, and a proof assistant requires locality. The third is that the community incentive structure rewarded ergonomics over rigor, with the result that languages designed to reach mass adoption invested in surface polish at the cost of the very features that would have made mechanization feasible.

The first two reasons are direct consequences of the human-author constraint. Removing that constraint, as T1 proposes, removes the obstacle. The third reason is a consequence of who languages were being adopted by (humans), and so dissolves at the same time. The reasons mechanized metatheory was infeasible were the same reasons human authorship was paramount. When the latter changes, the former changes with it.

Two things have happened in parallel with the shift in authorship. First, proof assistants have matured. The current generation (Moura and Ullrich 2021) supports, for the first time, the kind of dependently-typed, modular, refactor-friendly proof development that a language metatheory requires. The infrastructure for managing proofs at the scale of a language definition exists, has been validated on production-grade artifacts including reimplementations of the proof assistant's own kernel (Carneiro 2024), and continues to improve. Second, the methodology has matured. There is now a recognized literature of how to structure a metatheory development: how to organize substitution lemmas, how to handle inductive evaluation contexts, how to bridge specification and verification layers, how to maintain a development across refactors of the language definition (Wadler 2015; Atkey 2018). The POPLmark Challenge (Aydemir et al. 2005) established two decades ago that benchmark mechanizations of non-trivial language metatheory are achievable; the intervening years have closed the cost gap between the benchmark and routine engineering. This methodology is itself the engineering output of two decades of research, and it is now available as a guide rather than as a frontier.

The combination of a smaller language to mechanize, a more capable tool to mechanize it with, and a known methodology for doing so moves mechanized metatheory from the research column into the engineering column. The work remains nontrivial, but it has moved into the budgetable range. Substantial mechanized results from inside the same proof-assistant ecosystem now appear regularly enough that the budget is no longer a question of feasibility but of allocation. Examples include garbage-free reference counting with full soundness proofs (Reinking et al. 2021) and LLM-assisted mechanization of non-trivial compiler-correctness proofs in roughly four days of agent time (Paraskevopoulou 2026).

For the machine that writes the code, mechanized metatheory means that the static guarantees of the language are not aspirational. The compiler accepts a program if and only if the program satisfies the typing rules; the typing rules are, in turn, proven to ensure absence of whole classes of runtime error. The author, the machine, can therefore rely on those guarantees as guarantees, not as defaults that occasionally fail in corner cases discovered after deployment. Whole categories of testing become unnecessary, not because they have been replaced by something equivalent, but because the property they were testing is now a theorem. A machine generating one million programs in a verified-foundation language has a different reliability profile in kind from a machine generating one million programs in an unverified-foundation language, and the difference compounds with scale.

For the human auditor (the occasional reader of §3), mechanized metatheory provides something else: a clear separation between the language's claims and the program's claims. If the language is unsound, no amount of program-level verification rescues programs written in it. If the language is sound, program-level verification has a stable foundation to build on. Mechanized metatheory makes this separation sharp: the language's claims have been audited by a proof assistant, once, definitively. The auditor's remaining work is purely program-level, which is the work the auditor was hired to do.

The objection: "this is still too expensive." The objection, against an unspecified concept of "expensive," is hard to engage with directly. We engage with the concrete form: that mechanized metatheory costs more, in person-time and calendar-time, than ordinary language engineering. We grant that it does. Concrete data on the magnitude is available: the seL4 microkernel verification reported roughly 25 person-years of proof effort against 2 person-years of implementation effort for a kernel of ~8,700 lines of C (Klein et al. 2009), establishing a baseline before the recent LLM-assisted methodology; a 2026 experience report mechanized a non-trivial CertiCoq compiler-correctness proof in roughly 96 hours of agent time using a contemporary code-and-proof agent (Paraskevopoulou 2026). The ratio between baseline cost and current cost is the trajectory the objection has to engage with. Independent of mechanization specifically, recent work on the economic structure of AI-augmented software development reaches a compatible conclusion: specification discipline rather than raw model capability is the binding constraint on dependability, and the cost-benefit calculus favours up-front specification governance over after-the-fact remediation (Farrag 2026). The ratio has shrunk to the point where it is acceptable, and matters less than the multiplier the investment carries. Mechanized metatheory is paid once, per language. Its benefits are reaped per-program, by every program ever written in that language. The investment amortizes, at scale, over a number of programs that for a language whose surface is generated rather than authored is itself effectively unbounded. Comparing the one-time cost against the lifetime aggregate benefit is the only comparison that respects the structure of the investment, and on that comparison mechanized metatheory is not expensive.

A calibration of ambition is in order. T2 does not claim that *every* property of a language must be mechanized. It claims that the core (the type system, the operational semantics, the principal safety theorems) must be. Beyond that core, properties may be claimed informally, tested empirically, or proven on a per-program basis. The point of T2 is that the core is the part on which everything else depends, and has, until now, been treated as too expensive to fully verify. That excuse no longer holds, and any new language for which it is invoked is, in this sense, a language designed for the wrong author.

5 The Third Claim: The Pipeline Must Be a Formal Object (T3)

With a small surface granted (T1) and that surface verified (T2), the chain that runs through the surface, from human intent down to code, and from code down to executing target, is what remains to be addressed.

We claim that, given T1 and T2, this chain is the next obstacle to a programming language whose claims are reliable end-to-end, and that compilation must therefore be promoted from an engineering artifact to a formal object at every stage, not only at the once-traditional source-to-target step.

A compilation chain, in mainstream practice, runs from human-authored source code through a sequence of intermediate forms to machine code. Each step is a program that reads one form and produces another; the correctness of each is, in the dominant case, established by testing. Narrow per-compilation validation has been deployed for selected optimization passes in production toolchains (Lopes et al. 2021), but it does not extend to the structural translations between intermediate representations and is not the regime under which most compilation runs. The methodology has produced compilers of considerable engineering quality, and we are not arguing against the engineering. We are arguing that, in the new design center, the chain has grown an additional stage at the top (synthesis from intent), and this stage is presently treated by an entirely different methodology (statistical generation, training-data validation, post-hoc review) which inherits even fewer correctness guarantees than the engineering tradition that sits below it. The chain as a whole has become longer and weaker.

When the human was the author of code, the chain of trust ran: human writes source, compiler translates, machine runs. Two of the three links were highly trusted. The human author could be cross-examined; the executing machine's hardware-level correctness was orders of magnitude better than the layers above it. The compiler was the weakest link, but its bugs were typically discovered, because human authors notice when their code does not do what they wrote.

The new chain runs longer: human writes intent, system synthesizes code, compiler translates, machine runs. The chain has grown a stage at the top, and the social process which historically masked bugs has been weakened at two points. The human is no longer at the source of the chain (they are *one stage above* the source), and the historical bug-detection mechanism (the human notices that the code does not do what they wrote) requires the human to read the synthesized code, which is precisely what the new design center expects them not to do. The synthesizer, like the compiler, is a translator; like the compiler, it has bugs; unlike the compiler, it is at present subject to no correctness regime that can rule out whole classes of bug. Whatever errors are injected at the synthesis stage are unlikely to be caught downstream, because there is no downstream party with the knowledge or the practice of catching them.

This is a structural change. It does not say the chain has more bugs than before; it says that the social process which historically masked translation bugs no longer masks them, and that the chain has grown a stage which has *never* had such a social process and now never will. Translation correctness, which was a comfort at one stage and absent at another, is becoming a load-bearing requirement at every stage.

A compilation pipeline meeting the standard T3 calls for must, at every stage, satisfy one of two conditions. Either the transformation is *proven correct as a meta-theorem* (once, for all inputs, with the proof checkable by the same proof-assistant infrastructure that mechanized the language semantics (Leroy 2009; Kumar et al. 2014)), or it is *certified per-program*, with each compilation producing a proof-carrying artifact that the specific output preserves the semantics of the specific input (Necula 1997; Pnueli et al. 1998). Neither is unprecedented. Both have been demonstrated. T3 claims they should be combined into the normal case.

Verified compilation has been demonstrated for source languages substantially larger than what T1 admits (Leroy 2009; Kumar et al. 2014). The achievement is older than this paper and we do not recapitulate it. T1's contribution is to make it cheaper. A compilation pipeline whose source language has a small, regular surface admits per-stage formal treatment in a way a pipeline with a large, irregular source language has

never permitted. The cost of T3 is not the cost of verifying compilation in general; it is the cost of verifying compilation of a language designed to make verification tractable. The supporting infrastructure, modern Satisfiability Modulo Theories (SMT) solvers (Moura and Bjørner 2008) and frameworks for systems-level verification (Lattuada et al. 2024), is now mature enough that the bottleneck is no longer tooling. The cost is achievable, in the present moment, as ordinary engineering. We claim it is therefore the new floor, and that any pipeline below it is, in the new design center, a pipeline that falls short of what is now achievable.

The objection: "we already have testing." The objection treats testing as a substitute for proof, which it has never been: the limitation that testing can establish the presence of bugs but not their absence is canonical (Dijkstra 1972), and the limitation tightens in the LLM-application setting, where a recent survey identifies four structural gaps that make standard software-testing methodology inadequate to LLM-generated artifacts (Ma et al. 2025). The point at which the difference became consequential is precisely the point at which code stopped being authored and started being synthesized. Testing finds bugs that someone thought of. Proof rules out bug classes that nobody thought of. For a population of programs of bounded size, bugs that someone thought of dominate; testing was therefore approximately correct as a trust mechanism. For a population of programs of unbounded size (which is the population a synthesis system produces), bugs nobody thought of begin to dominate. Testing, on that population, is sampling from a distribution whose tail it cannot see. Proof is the principal tool that addresses tail risk at scale, and tail risk at scale is the failure mode of code that no human reads before it runs. T3 is the requirement that translation correctness, at every stage of the chain, be addressed by the tool that scales, not the tool that does not.

One observation closes T3. T1, T2, and T3 are not three independent claims; they are a system of mutually-supporting claims. T1 grants a small surface. T2 verifies the surface. T3 verifies the chain that runs through the surface, from intent above to machine below. Together, they describe a programming language in which the correctness of an emitted program (synthesized from human intent, lowered through code to machine instructions, executed by hardware) is, end to end, a formal artifact. Each claim alone is achievable in isolation and has been achieved. What the new design center makes available is their joint achievement, and what this paper argues is that the joint achievement should be the norm.

6 What This Implies for the Field

The thesis is operationalized in §3 through §5. §6 turns to the field at large: what should change in how programming languages are designed, taught, and researched, given that the design center has moved.

A natural objection to P1–P5 (Naur 1985) holds that the theory of a program (its rationale, its mapping to the world, its modification guidance) is irreducibly tacit and resides in the human authoring team. We acknowledge the objection and reframe it. In the new architecture the theory does not vanish; it migrates one level up. The human holds the theory of the *intents*. The code-level artifact, regenerable on demand from intent, no longer needs to carry the theory the way Naur's program text did. Naur's "program death" problem dissolves at the code layer because the canonical artifact is no longer the code. This relocation of the theory shapes the community recommendations that follow.

We address four communities, in rough order of immediacy.

Language designers. The standing default has been to inherit features from the lineage of mainstream languages and to justify only the deviations. We argue for the opposite default. Every feature inherited from the human-author era should now be a deliberate choice, not an inheritance. The question is no longer "what other features does this language need to be competitive?" The question is "does this feature serve the actual

author?" and the actual author is no longer guaranteed to be human. Many features will survive the audit; many will not. The exercise of conducting the audit is itself the contribution we ask of language designers, because it is the exercise the field rarely conducts from first principles, and has not conducted on this scale since the first generations of high-level languages were designed.

Verification researchers. Mechanized metatheory and verified compilation have lived as research specialties for thirty years. The arguments for keeping them in that category were arguments about cost: the techniques were valuable in the way difficult research artifacts are valuable, and the cost-to-deploy ratio left them outside the engineering mainstream. T1 collapses the cost; T2 generalizes the benefit. The verification community should expect and prepare for the transition of its tools from research artifacts to engineering defaults. The literature is ready for this transition. The community attitude is not always ready, because the success metric of a research field is novelty, and a tool that becomes a default is no longer novel. The community's metric for impact, in this transitional period, should weight adoption alongside novelty rather than separately from it. A natural new research target that follows directly from the recentering is *verified synthesis*: the formal certification of the intent-to-code stage as a meta-theorem, parallel to what verified compilation is for the source-to-target stage. Early work in this direction is beginning to appear, including spec-synthesis pipelines that emit Lean specifications with traceable clause-level refinement (Ye et al. 2026) and multi-modal verifiers that decompose end-to-end synthesis into independently checkable stages (Feng et al. 2026). The literature does not yet substantially address verified synthesis as a category; we argue here that it should.

AI-tooling researchers. A substantial body of current work is concerned with making existing human-designed languages tractable for machine generation: better completion models, better error correction, better post-hoc verification. This is reasonable work for the transitional period, and we do not argue against it. We argue that the long-term value lies elsewhere: in languages designed for the destination, where the language's surface is no longer a target for human-readable generation but an internal compilation step between intent and machine. This argument is consonant with a recent reframing of the foundational question, articulated in interview form by Heule (2025), who argues that verification-based trust should replace human comprehension as the trust anchor for machine-generated artifacts. The languages the AI-tooling community is spending effort to fit their tools to may, in their final form, not be the right languages at all. Designing languages whose surface is treated as IR rather than as user interface is itself a research program, parallel to and complementary to the research program of fitting machines to existing languages. Both are necessary now; only one is necessary in twenty years.

Educators and curriculum designers. The reframing of the human as occasional auditor rather than primary author is the natural continuation, under new conditions, of the disciplined modesty articulated by Dijkstra (1972): the programmer who knows what they cannot reliably do builds with techniques that compensate for cognitive limits and refuses techniques that compound them. We note the contrary tradition, most prominently Knuth (1984), which treats code as a literary artifact written primarily for human readers; we claim that the literate document is now the intent specification, not the synthesized code, and Knuth's argument migrates one level up rather than dissolving. The question "which language do I teach?" has, until now, had a stable answer: the language that gives the student the best ergonomic experience while learning the discipline. That answer is, for the student preparing to express intent above the code level and to audit synthesized code below it, no longer adequate. The student needs to learn to read, audit, and reason about code written in languages whose surface is treated as IR rather than as user interface, which means the ergonomic experience, while still relevant, is no longer the dominant criterion. The transition will be gradual, and educators reasonably remain cautious about premature commitment to languages that have not stabilized. The transition has, however, begun, and the curricula that anticipate it will produce graduates better positioned

for the workforce they will enter.

The four communities act on different timescales. Language designers and verification researchers can begin work immediately, and have begun. AI-tooling researchers will follow as the languages mature. Educators will follow last, as they should; curricula respond to industry rather than the other way around. The order is not a criticism of any of the communities; it is an observation about how change propagates through technical fields, and the propagation is already in progress.

7 Conclusion

The constituency that programming language design has historically optimized for, the human author of code, is dissolving. The field should change. We have argued that it should change in a specific direction: a deliberately small surface, because surface ergonomics is now a cost rather than a benefit (T1); a fully mechanized static semantics, because the verification floor has moved upward (T2); a compilation chain (from intent through code to machine) that is end-to-end a formal object, because the chain of trust no longer terminates in any party that can audit informally (T3). The three claims are joint. Each is individually achievable and has been individually achieved. The combination is what the new design center makes practical, and what the recentering of authorship makes necessary.

The premise (P1–P5) carries the argument's weight. If the human author of code is being displaced upward to intent and the synthesizer of code is best understood as a translation step, the conclusions follow. The research program is then to make each stage of the new chain (intent specification, synthesis, language metatheory, compilation) a verified object, with verified synthesis as the still-open frontier above the established craft of verified compilation.

References

Atkey, Robert. 2018. "Syntax and Semantics of Quantitative Type Theory." *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*, 56–65. https://doi.org/10.1145/3209108.3209186.

Aydemir, Brian E., Aaron Bohannon, Matthew Fairbairn, et al. 2005. "Mechanized Metatheory for the Masses: The POPLmark Challenge." *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, LNCS, vol. 3603: 50–65. https://doi.org/10.1007/11541868_4.

Brooks, Frederick P. 1987. "No Silver Bullet — Essence and Accidents of Software Engineering." *IEEE Computer* 20 (4): 10–19. https://doi.org/10.1109/MC.1987.1663532.

Carneiro, Mario. 2024. *Lean4Lean: A Reimplementation of the Lean 4 Kernel in Pure Lean 4*. https://github.com/digama0/lean4lean.

Cedar Policy team. 2026. *Cedar-Spec: Lean 4 Formalization and Differential Random Testing for Cedar*. https://github.com/cedar-policy/cedar-spec.

Congard, Sidney, Guillaume Munch-Maccagnoni, and Rémi Douence. 2025. *Linear Effects, Exceptions,*

and Resource Safety: A Curry–Howard Correspondence for Destructors. https://doi.org/10.48550/arXiv.2510.23517.

De La Cruz, Elyson. 2026. *From Code-Centric to Intent-Centric Software Engineering: A Reflexive Thematic Analysis of Generative AI, Agentic Systems, and Engineering Accountability*. https://doi.org/10.48550/arXiv.2605.11027.

Dijkstra, Edsger W. 1972. "The Humble Programmer." *Communications of the ACM* 15 (10): 859–66. https://doi.org/10.1145/355604.361591.

Dijkstra, Edsger W. 1978. *On the Foolishness of "Natural Language Programming"*. EWD 667. Available at https://www.cs.utexas.edu/~EWD/transcriptions/EWD06xx/EWD667.html.

Farrag, Sabry E. 2026. *The Productivity-Reliability Paradox: Specification-Driven Governance for AI-Augmented Software Development*. https://doi.org/10.48550/arXiv.2605.01160.

Fei, Haoxiang, Yu Zhang, Hongbo Zhang, Yanlin Wang, and Qing Liu. 2024. "MoonBit: Explore the Design of an AI-Friendly Programming Language." *Proceedings of the 1st ACM International Conference on AI-Powered Software (LLM4Code Workshop at ICSE 2024)*. https://doi.org/10.1145/3643795.3648376.

Feng, Yueyang, Dipesh Kafle, Vladimir Gladshtein, et al. 2026. *Certified Program Synthesis with a Multi-Modal Verifier*. https://doi.org/10.48550/arXiv.2604.16584.

Floyd, Robert W. 1967. "Assigning Meanings to Programs." *Proceedings of Symposia in Applied Mathematics, Vol. 19: Mathematical Aspects of Computer Science*, 19–32.

Gurgul, Vincent, Robin Gubela, and Stefan Lessmann. 2025. *The State of Generative AI in Software Development: Insights from Literature and a Developer Survey*. https://doi.org/10.48550/arXiv.2603.16975.

Haynes, Houston. 2026. *Dimensional Type Systems and Deterministic Memory Management: Design-Time Semantic Preservation in Native Compilation*. https://doi.org/10.48550/arXiv.2603.16437.

He, Kaifeng, Xiaojun Zhang, Peiliang Cai, et al. 2026. *Bridging Generation and Training: A Systematic Review of Quality Issues in LLMs for Code*. https://doi.org/10.48550/arXiv.2605.05267.

Heule, Marijn. 2025. *To Have Machines Make Math Proofs, Turn Them into a Puzzle*. Interview with J. Pavlus, *Quanta Magazine*. https://www.quantamagazine.org/to-have-machines-make-math-proofs-turn-them-into-a-puzzle-20251110/.

Hoare, C. A. R. 1973. *Hints on Programming Language Design*. STAN-CS-73-403. Stanford University Computer Science Department.

iMatix Corporation. 1995. *GSL: Generator Scripting Language*. https://github.com/imatix/gsl.

Klein, Gerwin, Kevin Elphinstone, Gernot Heiser, et al. 2009. "seL4: Formal Verification of an OS Kernel."

Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP '09), 207–20. https://doi.org/10.1145/1629575.1629596.

Knuth, Donald E. 1984. "Literate Programming." *The Computer Journal* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.

Kumar, Ramana, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. "CakeML: A Verified Implementation of ML." *Proceedings of POPL 2014*, 179–91. https://doi.org/10.1145/2535838.2535841.

Lattuada, Andrea, Travis Hance, Chanhee Cho, et al. 2024. "Verus: Verifying Rust Programs Using Linear Ghost Types." *Proceedings of the ACM on Programming Languages* 8 (OOPSLA1). https://doi.org/10.1145/3649833.

Leroy, Xavier. 2009. "Formal Verification of a Realistic Compiler." *Communications of the ACM* 52 (7): 107–15. https://doi.org/10.1145/1538788.1538814.

Lopes, Nuno P., Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. "Alive2: Bounded Translation Validation for LLVM." *Proceedings of PLDI 2021*, 65–79. https://doi.org/10.1145/3453483.3454030.

Ma, Wei, Yixiao Yang, Qiang Hu, et al. 2025. *Rethinking Testing for LLM Applications: Characteristics, Challenges, and a Lightweight Interaction Protocol*. https://doi.org/10.48550/arXiv.2508.20737.

Martin-Löf, Per. 1984. *Intuitionistic Type Theory*. Studies in Proof Theory, Lecture Notes 1. Bibliopolis.

Mell, Stephen, Botong Zhang, David Mell, et al. 2025. *Quasar: A Fast, Reliable, and Secure Programming Language for LLM Agents with Code Actions*. https://doi.org/10.48550/arXiv.2506.12202.

Moura, Leonardo de. 2026. *When AI Writes the World's Software, Who Verifies It?* https://leodemoura.github.io/blog/2026/02/28/when-ai-writes-the-worlds-software.html.

Moura, Leonardo de, and Nikolaj Bjørner. 2008. "Z3: An Efficient SMT Solver." *Proceedings of TACAS 2008*, LNCS, vol. 4963: 337–40. https://doi.org/10.1007/978-3-540-78800-3_24.

Moura, Leonardo de, and Sebastian Ullrich. 2021. "The Lean 4 Theorem Prover and Programming Language." *Proceedings of CADE-28*, LNCS, vol. 12699: 625–35. https://doi.org/10.1007/978-3-030-79876-5_37.

Naur, Peter. 1985. "Programming as Theory Building." *Microprocessing and Microprogramming* 15 (5): 253–61. https://doi.org/10.1016/0165-6074(85)90032-8.

Necula, George C. 1997. "Proof-Carrying Code." *Proceedings of POPL 1997*, 106–19. https://doi.org/10.1145/263699.263712.

Nian, Yi, Aojie Yuan, Haiyue Zhang, Jiate Li, and Yue Zhao. 2026. *Auditable Agents*. https://doi.org/10.

48550/arXiv.2604.05485.

Paraskevopoulou, Zoe. 2026. *Machine-Generated, Machine-Checked Proofs for a Verified Compiler (Experience Report)*. https://doi.org/10.48550/arXiv.2602.20082.

Pierce, Benjamin C. 2002. *Types and Programming Languages*. MIT Press.

Pnueli, Amir, Michael Siegel, and Eli Singerman. 1998. "Translation Validation." *Proceedings of TACAS 1998*, LNCS, vol. 1384: 151–66. https://doi.org/10.1007/BFb0054170.

Reinking, Alex, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. "Perceus: Garbage Free Reference Counting with Reuse." *Proceedings of PLDI 2021*, 96–111. https://doi.org/10.1145/3453483.3454032.

Soltanian Fard Jahromi, Ali, Amjed Tahir, Peng Liang, and Foutse Khomh. 2026. *On Fixing Insecure AI-Generated Code Through Model Fine-Tuning and Prompting Strategies*. https://doi.org/10.48550/arXiv.2605.05867.

Thompson, Ken. 1984. "Reflections on Trusting Trust." *Communications of the ACM* 27 (8): 761–63. https://doi.org/10.1145/358198.358210.

Töpfer, Michal, František Plášil, Tomáš Bureš, and Petr Hnětynka. 2026. *Vibe-Coding: Feedback-Based Automated Verification with No Human Code Inspection, a Feasibility Study*. https://doi.org/10.48550/arXiv.2604.14867.

Trooskens, Geert, Aaron Karlsberg, Anmol Sharma, et al. 2026. *Compiled AI: Deterministic Code Generation for LLM-Based Workflow Automation*. https://doi.org/10.48550/arXiv.2604.05150.

Ustynov, Dmytro. 2026. *Beyond Human-Readable: Rethinking Software Engineering Conventions for the Agentic Development Era*. https://doi.org/10.48550/arXiv.2604.07502.

Vibe Coding Needs Vibe Reasoning. 2025. https://doi.org/10.48550/arXiv.2511.00202.

Wadler, Philip. 2015. "Propositions as Types." *Communications of the ACM* 58 (12): 75–84. https://doi.org/10.1145/2851499.

Wirth, Niklaus. 1995. "A Plea for Lean Software." *IEEE Computer* 28 (2): 64–68. https://doi.org/10.1109/2.348001.

Ye, Zhe, Aidan Z. H. Yang, Huangyuan Su, et al. 2026. *VeriSpecGen: Intent-Aligned Formal Specification Synthesis via Traceable Refinement*. https://doi.org/10.48550/arXiv.2604.10392.