hexagon:2611.00323v1

# Expressiveness, Verifiability, Tractability: Toward a Computational Thermodynamics

Benjamin Kaminsky*

November 11, 2026

**Abstract**

An investigation into the existence of a conservation law governing computational behaviour, in the form of the Expressiveness, Verifiability, Tractability (EVT) Hypothesis, is undertaken. The hypothesis is motivated by a simple observation: mechanisms that increase what a system can directly express, powerful control, effects, reflection, meta-programming, and self-interpretation, tend to undermine what can be verified uniformly and mechanically, while regimes that maximize static guarantees typically do so by restricting expressiveness or relocating effort into compilers, proof artifacts, or human interpretation. We analyze canonical failure modes that arise when one attempts to push beyond this frontier, distinguishing collapse of correctness, collapse of compilability, and collapse of comprehensibility. We then develop an order-theoretic model for comparing languages and verification disciplines and define an "E/V boundary" as a Pareto frontier in a partially ordered design space. Finally, we sketch how analogous tradeoffs appear in other computational domains, suggesting that "thermodynamic" boundary behaviour may be a recurring feature of expressive formalisms that demand strong global guarantees and within physical methods of information processing.

**Keywords:** computation, programming languages, EVT hypothesis, resource theory

*benkaminskymath@gmail.com

**Contents**


**1 Introduction** **3**

**2 Expressiveness, Verifiability, Tractability** **4**
2.1 The EVT Hypothesis . . . 4
2.2 Expressiveness . . . 7
2.3 Verifiability . . . 10
2.4 Tractability . . . 12
2.5 Macro-expression and Rice-genericity . . . 14

**3 EVT: Scenarios and Failures** **15**
3.1 Gödel's Type System . . . 16
3.2 Tarski's Compiler . . . 18
3.3 Rice's Revenge . . . 19
3.4 Landin's Ghost? . . . 23

**4 Modeling Expressiveness vs. Verifiability Tradeoffs** **25**
4.1 Computation of E and V . . . 25
4.2 The E- and V-orders . . . 25
4.3 Endpoints: Lambda-calculus as Calibration . . . 26
4.4 The E/V Frontier as a Pareto Boundary in a Poset . . . 26
4.5 A Programming Language Design Space . . . 28
4.6 The Turing Tarpit and Real-life Programming Languages . . . 29
4.7 Attack Vectors Against the EVT . . . 30

**5 Toward a Computational Thermodynamics** **32**
5.1 Lattice Gauge Theory . . . 32
5.2 Numerics and PDEs . . . 33
5.3 Many-body Systems and Field Theory . . . 33
5.4 Resource Theory and the Future of the EVT Hypothesis . . . 34

**A Disclosures** **36**

## 1 Introduction

> *"ISWIM can be looked on as an attempt to deliver LISP from its eponymous commitment to lists, its reputation for hand-to-mouth storage allocation, the hardware-dependent flavor of its pedagogy, its heavy bracketing, and its compromises with tradition..."*
>
> — Peter Landin, *The Next 700 Programming Languages* (1966) [Lan66]

What are the physical limits of how computational processes can behave? Can we make the *perfect* programming language?

Programming languages are often introduced as notational conveniences for describing the same underlying class of computable functions. That perspective is correct but incomplete. In practice, languages differ sharply in *runtime behavior*: evaluation strategy, control flow, resource usage, effect discipline, and the kinds of invariants that can be expressed (or enforced) compositionally. These differences are not cosmetic. They determine what programmers can feasibly build, what compilers can safely optimize, and what properties can be verified without collapsing into undecidable or intractable reasoning.

Three long-running research programs frame this paper.

**Expressiveness via macros.** Landin's program is a canonical starting point: rather than treating languages as a zoo of ad hoc syntactic devices, he pushes toward a small set of semantic primitives that generate families of languages with principled behavior [Lan66]. Later, Felleisen makes the informal folklore of "language $A$ is more expressive than language $B$" precise by giving a notion of expressiveness grounded in definability/translation criteria [Fel91]. This line naturally connects to the semantics tradition in which observable program behavior is characterized operationally and compared to denotational models; in particular, full abstraction and game semantics give a mature account of when a semantic model captures exactly what programs can observe about one another [AJM00]. Meanwhile, the study of effects and control reveals that small changes in expressive mechanisms (e.g., delimited continuations) can radically alter what can be encoded modularly, and what program transformations remain valid [DPJS07].

**Verifiability via types.** In parallel, type systems provide a spectrum of partial "static truth": from practical polymorphism and principled inference [Mil78], through foundational dependent type theories in which propositions are types and programs are proofs [CH88], to modern synthesized accounts that connect these ideas back to everyday language design [Pie02]. Yet types do not magically eliminate limits: they relocate them. Strengthening a system to express richer properties tends to increase proof burden, complicate inference, or require more sophisticated metatheory.

**Tractability via understanding.** The third axis is *tractability*: the end-to-end feasibility of using a language as a human tool. Unlike expressiveness (rooted in semantic comparisons and eliminability) and verifiability (rooted in type-theoretic proof disciplines and soundness theorems), tractability is not usually packaged as a single canonical research program in programming languages. Classical complexity theory focuses on the hardness of *decision problems*, whereas our concern is the cost profile induced by a language and its tooling: elaboration/compilation, evaluation, and verification, together with structural proxies for human control. A nearby foundational tradition is *implicit computational complexity*, which designs languages and type disciplines that guarantee resource bounds (often polytime) by construction [DL22]; at the other extreme, *cost semantics* and resource-aware analyses make evaluation cost explicit and compositional, enabling machine-checkable accounting of runtime behavior [DLR15, HAH12]. In this paper, we adopt tractability in a weaker but more general sense: not "all programs run fast," but rather that the language-induced overhead of elaboration, evaluation,

and verification remains controllably bounded as a function of program size, and that the resulting control/proof structure remains amenable to local reasoning.

**The question.** This paper is about the joint boundary of these programs: the tension between *expressiveness* (what behaviors and abstractions a language can represent), *verifiability* (what semantic and safety properties can be checked or proved with usable effort), and *tractability* (what remains computationally and socially feasible at scale). We use the acronym EVT for this triangle of constraints, and we treat it as a kind of "programming thermodynamics": not a vague metaphor, but a claim that language design lives under hard tradeoffs induced by computability, complexity, and the structure of semantic observation.

At a high level, the paper develops a framework for making EVT tradeoffs explicit and comparable across languages, and it identifies failure modes for hypothetical "maximal" languages that attempt to optimize all axes simultaneously. Concretely:

- We formalize a working notion of expressiveness and verifiability sufficient to compare languages beyond "they compute the same functions," building on established translation- and observation-based approaches [Fel91, AJM00].
- We isolate a small set of structural constraints (soundness, tractability, and human comprehensibility) and show how maximizing both expressiveness and verifiability forces at least one of these to fail in identifiable ways (the EVT failure trichotomy).
- We relate the resulting boundary to classical undecidability and to the practical realities of type systems and proof engineering [Mil78, CH88, Pie02].

**A motivating analogy: CAP.** A useful inspiration comes from distributed systems rather than programming language theory: the CAP theorem. In its standard form, CAP states that in the presence of network partitions, a distributed system cannot simultaneously guarantee strong *consistency* and *availability*; one must be weakened when partitions occur [GL02]. Although CAP is about fault-tolerance in a particular physical model (asynchrony and partitions), its rhetorical lesson generalizes: some desirable properties form an *incompatible triple* once one includes a realistic adversary (here, partitions) and demands nontrivial guarantees. We use CAP only as an analogy—not as a technical reduction—to motivate the stance of this paper: language design also lives under hard incompatibilities once one demands (a) rich behavioral expressiveness, (b) strong, usable verification principles, and (c) tractable evaluation and reasoning. The EVT framework can be read as a "CAP-like" tradeoff for languages: not the same theorem, but a similar claim that the limits are structural rather than merely historical or aesthetic.

## 2 Expressiveness, Verifiability, Tractability

### 2.1 The EVT Hypothesis

We state the main conjecture below.

**Conjecture 2.1** (Expressiveness/Verifiability/Tractability ("EVT") Hypothesis)**.** *The conjecture formally comprises two parts:*

*(A)* ***Loss of Global Verifiability in Macro-expressive Systems.*** *As the expressive power of a system of physical computation increases, the range of its globally algorithmically verifiable behaviors generally decreases unless additional structural restrictions are imposed.*

*(B)* ***Expressiveness/Verifiability/Tractability Trilemma.*** *Within such a system, every deterministic computational model is fundamentally constrained by a three-way tradeoff among* expressiveness *(geometric or semantic generality),* verifiability *(algebraic checkability or formal runtime assurance), and* tractability *(algorithmic or computational efficiency). These attributes satisfy the following trilemma:*

**No system can simultaneously maximize all three properties.** For any given model, at most two of the three can be optimized beyond a critical threshold, while the third must necessarily be sacrificed.

A selection principle and conservation law for computational behavior. First mentioned in the context of topological gauge theories in [Kam25].[1] For readers more comfortable with physical language, the trilemma can be re-expressed in standard thermodynamic language. These are *not* physical laws; they are organizing claims about structural limits induced by computability, complexity, and the interaction costs of programming and proof. Each law should be read relative to the definitions already fixed: macro-expressiveness (section 2.2), type-based verifiability (section 2.3), and polynomial-overhead tractability (section 2.4).

**Conjecture 2.2** (The First Law of Programming Thermodynamics)**.** *There exists a theoretical upper boundary in the feasible region of language designs: beyond that boundary, no language can simultaneously increase expressiveness and verifiability* while retaining *(a) soundness of its verification discipline, (b) compiler/pipeline tractability, and (c) human comprehensibility.*

**Interpretation.** The First Law is a geometric restatement of the EVT trilemma already discussed: if we view "language designs" as points in a space parameterized by (at least) expressiveness $E$, verifiability $V$, and tractability $T$, then the set of *viable* points is not all of $\mathbb{R}^3$ but a constrained region. The claim is that this region has a *nontrivial boundary*: pushing a design past it forces at least one of the three guardrails to fail.

**Operational reading.** Attempts to build a "maximal" language (high $E$ and high $V$) tend to fail in recognizable ways:

(F1) **Consistency failure.** The static discipline becomes unsound (the verifier certifies false semantic claims).

(F2) **Compilability failure.** Elaboration/typechecking/evaluation incur uncontrolled blowups (superpolynomial overhead, global proof search, etc.).

(F3) **Comprehensibility failure.** The system becomes unusable by humans except via opaque automation, defeating the point of a programming language as an *interface*.

We treat these as distinct "failure modes" rather than a single generic impossibility.

**Conjecture 2.3** (The Second Law of Programming Thermodynamics)**.** *In a sufficiently expressive regime, increasing verifiability induces an entropic loss in expressiveness, and vice versa: the tradeoff is directional and cannot, in general, be reversed without some form of external intervention (e.g. restricting the language, strengthening trusted automation/oracles, or importing additional proof burden).*

**Interpretation.** The Second Law asserts a *directional* coupling between $E$ and $V$ once expressiveness is above a threshold where nontrivial semantic phenomena appear (effects, control, higher-order state, reflection, etc.). In that regime, "turning up" verifiability typically means introducing restrictions, annotations, or proof obligations that reduce what can be expressed *directly* (macro-expressiveness), or at least reduce what can be expressed *tractably*.

**A minimal formal reading.** Fix a baseline core $L_{\mathsf{core}}$ and consider a family of surface languages $\{L_\alpha\}$ that share the same core evaluation model but differ by (a) added expressive facilities and (b) verification discipline. The Second Law can be read as the claim that, in such families, there is a persistent monotonic tension:

$$\Delta V > 0 \quad \Rightarrow \quad \Delta E < 0 \ \text{ or } \ \Delta T < 0,$$

and dually for $\Delta E > 0$. Here "entropic" is shorthand for the observation that once global semantic power is admitted, restoring strong verification typically requires adding structure (types/proofs/constraints) whose management creates irreversible overhead: proof artifacts accumulate, inference requires human guidance, or compilation must perform whole-program reasoning.

[1] "Professor, why is the Python code running slowly?" "Aha, well you see, the geometry of a Yang–Mills field is such that there are these computational helices and uh..."

**"Irreversible without intervention."** We do *not* mean that language evolution is literally irreversible, but that there is no general-purpose, purely local refactoring that simultaneously restores lost verifiability *and* retains full expressiveness at the same tractability level. Something must give:

- restrict the language (e.g. to total/terminating fragments),
- accept larger proof/certificate burdens,
- rely on stronger trusted automation (SMT, proof search, certified compilers),
- or weaken the property being verified.

This parallels the way CAP is used as an analogy: the adversary (semantic power, undecidability, interaction cost) forces a directional compromise.

**Conjecture 2.4** (The Third Law of Programming Thermodynamics)**.** *As verifiability approaches its theoretical maximum, expressiveness must approach triviality (e.g., total or strongly normalizing languages, or systems whose observational theory is decidable). However, no nondegenerate system of computation can reach zero expressive power.*

**Interpretation.** The Third Law isolates an "absolute zero" phenomenon: maximal verifiability—understood as a regime where semantic reasoning becomes uniformly decidable, proof obligations are bounded, and equivalence is mechanically tame—forces the language toward a class with sharply limited behaviors. In practice, this often means some combination of:

- **termination by design** (total functional programming/normalization-based proof assistants),
- **restricted effects/control** (to preserve decidable checking and tractable reasoning),
- **finite-state/decidable observational theories** (so verification is complete for a rich class of properties).

**Why "triviality" is the limit "zero expressiveness" is unattainable.** If we idealize "maximal verifiability" as demanding that a large class of semantic questions (including nontrivial equivalences or refinement checks) be decidable and compositional, then classical undecidability pressures force severe restrictions on behavior. Totality/strong normalization is one principled route: it makes definitional equality and proof checking tractable, but it also excludes many expressive phenomena that rely on general recursion or unrestricted effects.

A system with *literally* zero expressive power would admit no meaningful behavioral distinctions at all; its programs would collapse (up to contextual equivalence) into a single observational class or a finite set of trivial observations. Such a "language" is better viewed as a degenerate logic (e.g. a fragment whose evaluation is vacuous), not as a programming language. The Third Law, therefore, says: even in the limit where $V$ is pushed as high as possible, a viable system must retain some minimal degree of behavioral variation (at least enough to encode nontrivial propositions, data, or transformations), hence it cannot collapse to absolute expressive nullity.

The trilemma provides the local obstruction: in any sufficiently rich regime, one cannot maximize all desiderata simultaneously. The "laws" are a global packaging:

- the First Law asserts an upper boundary in the design space (a feasible region with edges),
- the Second Law asserts directionality of the $E$–$V$ tension in expressive regimes,
- the Third Law identifies the limiting behavior as $V$ approaches its maximum.

None of these claims requires us to commit yet to a unique numeric definition of $E, V, T$; they only require that our comparison notions behave monotonically under the standard moves of language design (adding primitives, restricting syntax/semantics, strengthening static disciplines, and shifting burden between the programmer and the machine).

### 2.2 Expressiveness

What is "expressive power"?

If we only ask which *functions* a language can compute, then (barring restrictions such as totality or strong normalization) most real languages collapse into the same equivalence class: they are universal. But programmers do not experience languages as mere function algebras. They experience *abstraction mechanisms*: binding forms, modules, control operators, effects, type disciplines, and the accompanying equational laws and reasoning principles. Expressiveness, in the sense relevant to language design, is about which abstractions can be provided *as primitives* versus which can be treated as *definitional* ("syntactic sugar") without disturbing the semantic fabric of the language.

The modern starting point for making this precise is Felleisen's program: compare languages not by the set of computable functions, but by the existence (or non-existence) of *structure-preserving eliminations* of proposed features [Fel91]. Informally, an extension is "inessential" if it can be compiled away by a local, syntax-directed translation that preserves the observable behavior of programs. This section sets up a framework that makes our *micro-* vs. *macro-* expressiveness distinction precise and robust enough to support later arguments about EVT tradeoffs.

Fix a *language universe* $\mathcal{L}$: a set of term-formers (constructors) together with a notion of *program* and an operational semantics. Concretely, one may think of $\mathcal{L}$ as a grammar of expressions $e$ built from a signature $\Sigma_{\mathcal{L}}$ together with:

- a subset $\mathrm{Prog}(\mathcal{L})$ of *closed* expressions designated as programs;
- a reduction relation $\to_{\mathcal{L}}$ (small-step) and/or an evaluation judgment $e \Downarrow_{\mathcal{L}} v$ (big-step);
- a set of observable outcomes Obs (e.g. termination with a value, raising an exception, producing an I/O trace, divergence).

A *sublanguage* $L \preceq \mathcal{L}$ is obtained by restricting $\Sigma_{\mathcal{L}}$ to a subsignature $\Sigma_L \subseteq \Sigma_{\mathcal{L}}$ and inheriting the induced semantics on those terms.

**Definition 2.1** (Conservative restriction/extension)**.** Let $L \preceq \mathcal{L}$ be a sublanguage. A further restriction $L_0 \preceq L$ is a *conservative restriction* if every $L_0$-program has the same observable behavior when evaluated in $L_0$ as when evaluated in $L$ (i.e. the added constructs of $L$ do not change the meaning of programs that do not use them). Dually, $L$ is a *conservative extension* of $L_0$ when $L_0 \preceq L$ and $L_0$ is a conservative restriction of $L$.

Conservativity is the minimum sanity condition needed to discuss whether "new" constructs add expressiveness: we do not want an extension to count as "more expressive" merely because it changes the semantics of the old language. To compare programs, we need a notion of observability. The most common choice is contextual observation: two terms are equivalent if no program context can tell them apart.

**Definition 2.2** (Program contexts and contextual equivalence)**.** A *program context* $C[\cdot]$ for a language $L$ is an $L$-expression with a single hole such that, whenever $e$ is an $L$-expression of the appropriate syntactic category, $C[e]$ is an $L$-program. We write $\mathrm{Ctx}(L)$ for the set of program contexts of $L$.

Fix an observation function $\mathrm{Obs}_L : \mathrm{Prog}(L) \to \mathrm{Obs}$ (e.g. Obs could distinguish termination-with-value from divergence, or could record traces). Two $L$-expressions $e, e'$ are *contextually equivalent*, written $e \approx_L e'$, if

$$\forall C[\cdot] \in \mathrm{Ctx}(L), \quad \mathrm{Obs}_L(C[e]) = \mathrm{Obs}_L(C[e']).$$

Note that

- The choice of Obs matters (may-termination vs. must-termination, trace equivalence, etc.). For the EVT story, we only need that $\approx_L$ is a congruence induced by contexts, and that it matches the intended notion of "same behavior" for the paper's examples.
- If one prefers, $\approx_L$ can be defined using a labeled transition system and trace sets; the development below still goes through.

**Eliminability and translations.** We now formalize what it means for an extension to be "just sugar." The key is to restrict the set of admissible translations. Let $L = L_0 + \{F_i\}_{i \in I}$ be a conservative extension of $L_0$ inside some universe $\mathcal{L}$, where the $F_i$ are the *additional* constructors (facilities) not present in $L_0$.

**Definition 2.3** (Eliminating translation)**.** An *eliminating translation* from $L$ to $L_0$ is a function

$$\phi : \mathrm{Expr}(L) \to \mathrm{Expr}(L_0)$$

satisfying the following conditions.

(E1) **Identity on the base.** $\phi$ is homomorphic for all constructors of $L_0$ (equivalently: it commutes with the formation rules of $L_0$), and on variables it acts as identity.

(E2) **Program preservation.** If $P \in \mathrm{Prog}(L)$ then $\phi(P) \in \mathrm{Prog}(L_0)$.

(E3) **Semantic preservation (adequacy).** For all programs $P \in \mathrm{Prog}(L)$,

$$\mathrm{Obs}_L(P) = \mathrm{Obs}_{L_0}(\phi(P)).$$

If such a $\phi$ exists, we say that the additional facilities $\{F_i\}_{i \in I}$ are *eliminable* (or *expressible*) in $L_0$ (with respect to $L$).

This notion is intentionally permissive: an eliminating compiler could, in principle, perform global rewrites that radically change term structure, as long as it preserves observable behavior.

**Macro-eliminability (structure-preserving eliminations).** Felleisen's key refinement is to capture the Landin intuition of *local* eliminations: translations that remove constructs *compositionally*, without inspecting whole-program structure. This matches how programmers and implementers actually treat "inessential" language features: they compile away by macro-expansion or fixed desugaring schemas [Fel91].

**Definition 2.4** (Syntactic abstractions)**.** A (multi-hole) *syntactic abstraction* over $L_0$ is a context-like term former

$$A(\cdot_1, \ldots, \cdot_k)$$

built from the constructors of $L_0$ with $k$ holes, such that plugging in $k$ expressions of the right categories yields an $L_0$-expression.

**Definition 2.5** (Macro-eliminability/macro-expressibility)**.** Let $L = L_0 + \{F_i\}_{i \in I}$ be a conservative extension. The added facilities are *macro-eliminable* if there exists an eliminating translation $\phi : L \to L_0$ such that, for each $a$-ary constructor $F \in \{F_i\}_{i \in I}$, there is a fixed $a$-ary syntactic abstraction $A_F$ over $L_0$ with

$$\phi\big(F(e_1, \ldots, e_a)\big) \;=\; A_F\big(\phi(e_1), \ldots, \phi(e_a)\big) \quad \text{for all } e_1, \ldots, e_a.$$

In this case we also say: $L_0$ *can macro-express the facilities* $\{F_i\}$ *(with respect to* $L$*).*

Intuitively: each new construct $F$ has a *single* expansion schema $A_F$ that expresses it using only $L_0$-constructs, and the translation of a compound phrase is obtained by composing the translations of its subphrases.

**Micro- vs. macro-expressiveness** The micro/macro distinction can now be aligned to the above.

**Definition 2.6** (Micro-expression (inessential extension))**.** A conservative extension $L = L_0 + \{F_i\}_{i \in I}$ is *micro-expressive* (relative to $L_0$, within $\mathcal{L}$) if the added facilities are macro-eliminable in the sense of definition 2.5. Equivalently: the extension admits a structure-preserving, local desugaring into $L_0$ that preserves program observations.

**Definition 2.7** (Macro-expression (essential extension))**.** A conservative extension $L = L_0 + \{F_i\}_{i \in I}$ is *macro-expressive* (relative to $L_0$, within $\mathcal{L}$) if the added facilities are *not* macro-eliminable; i.e. there exists at least one added constructor $F$ for which no semantics-preserving elimination can be given by a fixed syntactic abstraction schema over $L_0$.

Without restricting translations, almost any feature can be simulated by a sufficiently global compilation strategy (CPS conversion, whole-program defunctionalization, staging, proof-carrying code, etc.). Macro-eliminability is a principled way to prevent the comparison from collapsing, by requiring eliminations to be *local* and *structure-preserving* [Fel91]. This makes expressiveness claims reflect what language users typically mean: whether a feature can be provided as mere notation and library code without changing the underlying reasoning principles.

**Examples and non-examples.** The following examples calibrate the definitions. They are illustrative; later sections will pick a stable "reference universe" tailored to the EVT argument.

*Example* 2.1 (A canonical micro-expressive extension: `let`). In a call-by-value $\lambda$-calculus with functions, a `let`-binding is macro-eliminable by the standard desugaring

$$\phi(\texttt{let } x = v \texttt{ in } e) \;=\; (\lambda x.\phi(e))\,\phi(v),$$

with $\phi$ otherwise homomorphic. This is exactly Landin's paradigm of "inessential" constructs: a local compilation rule that preserves behavior.

*Example* 2.2 (A canonical "global" translation: CPS is not a macro translation). Continuation-passing style conversion provides a semantics-preserving translation from direct style into a continuation language, and conversely (under hypotheses) an inverse mapping. But CPS is a *whole-program* restructuring: it changes the shape of essentially every subterm, threads an extra continuation argument throughout, and typically requires global administrative reductions. Sabry and Felleisen's analysis of CPS reasoning highlights exactly this phenomenon: CPS-equivalent programs admit reasoning principles that are not available in the original direct-style calculus without additional axioms [SF93]. In our terminology, CPS exemplifies a powerful eliminating translation that is *not* macro-eliminability evidence.

*Example* 2.3 (Expressible but not macro-expressible: polymorphic `let` (phenomenon)). One of Felleisen's motivations for macro-expressiveness is that a feature may be eliminable in a permissive sense while failing to admit a local, structure-preserving elimination. A classic instance is that polymorphic `let` can be simulated in some monomorphic settings by global transformations, yet does not satisfy the macro-eliminability criterion in the relevant framework. This gap between "there exists an encoding" and "there exists a macro-expansion" is exactly what micro- vs. macro-expressiveness is meant to track.

**Relative expressiveness as a preorder.** Macro-eliminability yields a comparative relation between sublanguages inside a fixed universe.

**Definition 2.8** (Relative (macro-)expressiveness)**.** Let $L_0$ and $L_1$ be conservative restrictions of a common universe $\mathcal{L}$. We say that $L_1$ is *at least as macro-expressive as* $L_0$ (with respect to $\mathcal{L}$), and write

$$L_0 \preceq_{\text{macro}} L_1,$$

if whenever $L_0$ contains (or can macro-express) a set of additional $\mathcal{L}$-facilities, then $L_1$ also contains (or can macro-express) those facilities.

**Proposition 2.5.** *$\preceq_{\text{macro}}$ is a preorder on conservative restrictions of a fixed universe $\mathcal{L}$ (reflexive and transitive).*

*Proof sketch.* Reflexivity is immediate. Transitivity follows by composition of macro-eliminations: if $L_2$ macro-expresses what $L_1$ can macro-express, and $L_1$ macro-expresses what $L_0$ can macro-express, then $L_2$ macro-expresses what $L_0$ can macro-express. The technical point is that macro-eliminations compose because each eliminated construct is replaced by a fixed syntactic abstraction schema, hence the composite remains syntax-directed. □

The payoff of such definitions is that expressiveness becomes something we can *reason about* structurally:

- Micro-expressive extensions preserve the "semantic laws" of the core: they do not introduce fundamentally new observation principles, merely new surface forms.

- Macro-expressive extensions tend to invalidate or refine existing laws: new distinctions appear, new equivalences break, and reasoning about old code may require stronger semantic models or proof obligations [Fel91].

This "law invalidation" perspective is the bridge to EVT: increasing expressiveness is not free, because it changes what must be tracked to verify programs, and it changes which transformations remain sound.

### 2.3 Verifiability

What is *verifiability*?

Informally, it is the extent to which a language admits *effective, scalable methods* for establishing semantic properties of programs *before* they run. This includes safety properties ("does not go wrong"), representation invariants, effect-discipline claims ("no mutation occurs here"), and richer functional correctness assertions. The EVT perspective treats verifiability as a first-class design axis: languages are not just devices for *expressing* computations, but also for *proving* facts about those computations.

A first obstacle is that most interesting semantic properties of programs are not decidable in full generality. Thus, verifiability in practice almost always proceeds by (a) restricting the property language, (b) restricting the program language, or (c) accepting incomplete methods (semi-decision procedures, bounded model checking, interactive proofs, etc.). Type systems are the canonical instance of option (a): they provide a compositional, syntax-directed proof discipline whose judgments can usually be checked mechanically and locally. Pierce's presentation of type systems is particularly useful here because it isolates what we need for the EVT: a clean separation between *static semantics* (judgments and proofs) and *dynamic semantics* (evaluation), linked by a *soundness theorem* [Pie02].

To speak precisely, we model verification as a relation between programs and specifications.

**Definition 2.9** (Specifications and satisfaction)**.** Fix a language $L$ with programs $\mathrm{Prog}(L)$ and an observation map $\mathrm{Obs}_L : \mathrm{Prog}(L) \to \mathrm{Obs}$ (as in definition 2.2). A *specification* is an element $\varphi$ of some specification class $\Phi$ equipped with a satisfaction relation

$$P \models_L \varphi.$$

For example, $\varphi$ may assert safety (no stuck states), termination, trace constraints, noninterference, or refinement of an abstract model.

**Definition 2.10** (Verifier, soundness, completeness)**.** A (mechanical) *verifier* for $(L, \Phi)$ is a partial procedure

$$\mathcal{V} : \mathrm{Prog}(L) \times \Phi \rightharpoonup \{\mathsf{yes}, \mathsf{no}\}.$$

It is *sound* if whenever $\mathcal{V}(P, \varphi) = \mathsf{yes}$, we have $P \models_L \varphi$. It is *complete* if whenever $P \models_L \varphi$, we have $\mathcal{V}(P, \varphi) = \mathsf{yes}$.

In realistic settings, completeness is often too strong: we may settle for semi-decision procedures, or for soundness with false negatives, or for a proof-producing verifier that may require user guidance.

**Types as lightweight program logics.** Type systems instantiate verification by letting *types* themselves play the role of specifications. In the standard account (Pierce), a type system consists of a typing relation $\Gamma \vdash e : \tau$ together with an operational semantics for evaluation [Pie02]. The intended reading is:

$$\Gamma \vdash e : \tau \quad \Rightarrow \quad \text{"under assumptions } \Gamma\text{, } e \text{ behaves like a } \tau\text{-thing."}$$

What "behaves like" means depends on the semantics and the chosen notion of observation, but the central guarantee is *type safety*.

**Definition 2.11** (Type safety via progress and preservation (schema))**.** Let $L$ be a language with expressions $e$, values $v$, and evaluation step $e \to e'$. A type system for $L$ is *type safe* if it satisfies:

(TS1) **Preservation.** If $\Gamma \vdash e : \tau$ and $e \to e'$, then $\Gamma \vdash e' : \tau$.

(TS2) **Progress.** If $\varnothing \vdash e : \tau$ (closed well-typed term), then either $e$ is a value or there exists $e'$ with $e \to e'$.

This is the familiar Wright–Felleisen/Pierce pattern: well-typed programs do not get stuck, and evaluation preserves typing [WF94, Pie02]. In EVT terms, the typing judgment is a proof object (often implicit) that certifies a semantic property ("will not go wrong").

Crucially, types are *compositional* verifiers: judgments are derived by syntax-directed rules that mirror language structure. This compositionality is one of the primary reasons type systems scale: local proofs compose into global guarantees.

**The verifiability spectrum: from inference to proof engineering.** The *verifiability axis* is not binary. It ranges from fully automatic, low-friction verification to rich, proof-oriented systems with significant human effort. The tradeoff is familiar: the stronger the properties we want to express, the harder it becomes to infer, check, or construct proofs automatically.

**Automatic regimes.** At one end sit type disciplines designed for decidable checking and (often) decidable inference: e.g. Hindley–Milner style polymorphism and its descendants. The engineering aim is that the verifier is essentially invisible: a compiler elaborates implicit proofs and checks them cheaply [Mil78].

**Expressive regimes.** Moving outward, richer type systems allow richer invariants: polymorphism with higher-rank types, refinement-like constraints, linearity/effects, etc. These systems often preserve decidable *checking* but lose decidable *inference*, shifting work from the machine to the programmer (annotations, proofs, solver hints).

**Proof-carrying regimes.** At the far end are dependent type theories, where types may quantify over values and encode full specifications. Coquand and Huet's Calculus of Constructions is a canonical foundation: it unifies higher-order functions with a powerful dependent type discipline, enabling the Curry–Howard "programs as proofs" reading to become an engineering tool [CH88]. Here, the verifier is a type checker for proofs, but the proof objects may be large, and their construction may require substantial human guidance (tactics, automation frameworks, libraries). One can keep checking decidable by careful design (definitional equality via normalization), while accepting that *finding* proofs or inferring types is generally not algorithmically trivial.

From an EVT standpoint, this spectrum suggests a three-part decomposition of "verifiability":

(V1) **Spec expressivity.** How rich is the specification language $\Phi$ (what can be stated)?

(V2) **Mechanizability.** How much of the proof burden can be discharged automatically (checking vs. inference vs. synthesis)?

(V3) **Cost.** What is the computational and human cost of producing and checking certificates at scale?

We will treat these components as coupled: increasing (V1) tends to worsen (V2) or (V3), unless one restricts the underlying language, the logic, or the observation model.

**Working definition for the EVT.** We now isolate a notion of verifiability that is strong enough for the EVT boundary statements but weak enough to encompass both lightweight and proof-carrying type systems.

**Definition 2.12** (Type-based verifiability (EVT working notion))**.** Let $L$ be a language and let $\mathcal{T}$ be a type system for $L$ with a judgment

$$\Gamma \vdash_{\mathcal{T}} e : \tau$$

whose derivations are mechanically checkable. We say that $L$ is *type-verifiable up to* $\mathcal{T}$ if:

(TV1) **Checkability.** There is a total algorithm $\mathrm{Check}_{\mathcal{T}}$ that decides whether a purported derivation is a valid typing derivation in $\mathcal{T}$.

(TV2) **Soundness-to-semantics.** There is a semantic reading $[\![\tau]\!]$ of types as specifications such that well-typed closed programs satisfy the corresponding specification (type safety is the base case) [Pie02].

This is intentionally minimal: it does not require inference to be decidable, nor completeness with respect to an external specification language. It captures the core "verifiability move": convert semantic claims into checkable static artifacts, and prove a soundness theorem linking static proofs to dynamic behavior.

Later sections will sharpen this into the EVT tradeoff claims by (a) fixing specific classes of "semantic laws" that verifiers must preserve, and (b) distinguishing between verifiability-as-*decidable classification* and verifiability-as-*proof-carrying certification*. For now, the message is simple: *verifiability is the ability to attach enforceable, mechanically checkable meaning to programs*, and type systems are the dominant vehicle for doing so [CH88].

### 2.4 Tractability

What does it mean for a language, semantics, or verification pipeline to be *tractable*?

Unlike expressiveness and verifiability, "tractability" is not usually treated as a single unified research program in programming languages. Complexity theory, as traditionally formulated, studies the intrinsic difficulty of *decision problems* (e.g. **P** vs. **NP**), whereas our interest is broader: the *end-to-end cost* of using a language as a human tool. In particular, a language can be Turing-complete (hence able to express computations of arbitrarily large intrinsic runtime) while still being tractable in the sense that the *overhead induced by the language and its verification discipline* remains controlled.

There *is* a nearby line of work worth flagging: *implicit computational complexity* (ICC) studies how language restrictions and typing/logic disciplines can *guarantee* that all definable programs lie in a complexity class such as polynomial time [DL22]. That program is relevant to the EVT as an extreme point: it achieves strong tractability by deliberately sacrificing expressiveness. Our tractability axis is weaker and more pragmatic: it measures whether the *meta-processes* of elaboration, evaluation, and verification are feasible, and whether their costs are stable under realistic programming practice.

**Tractability as pipeline cost.** We model the use of a language as a pipeline:

$$\text{surface program} \Rightarrow \text{elaboration}, \quad \text{core program} \Rightarrow \text{evaluation}, \quad \text{observable behavior},$$

optionally accompanied by a proof/certificate that is checked by a verifier. To avoid baking in a particular compiler or typechecker, we define tractability relative to an explicit choice of (a) an elaboration system, (b) a costed operational semantics, and (c) a certificate checker.

**Definition 2.13** (Program size)**.** Fix an encoding of programs as finite strings. For a program $P$, write $|P|$ for its length (or any equivalent size measure, e.g. AST size).

**Elaboration cost.** Let $L_{\mathsf{surf}}$ be a surface language and $L_{\mathsf{core}}$ a conservative core calculus (often smaller, closer to evaluation). Elaboration includes macro-expansion, desugaring, and compilation into core.

**Definition 2.14** (Elaboration system with cost)**.** An *elaboration system* is a (partial) relation

$$P \Downarrow^{c_{\mathsf{elab}}} Q$$

where $P \in \mathrm{Prog}(L_{\mathsf{surf}})$, $Q \in \mathrm{Prog}(L_{\mathsf{core}})$, and $c_{\mathsf{elab}} \in \mathbb{N}$ is an elaboration cost. We write $\mathrm{Elab}(P) = Q$ when the witness cost is irrelevant.

Intuition: macro systems that allow small $P$ to expand into exponentially large $Q$ are *elaboration-intractable* in the sense relevant to the EVT, even though they may increase expressiveness.

**Evaluation cost.** To speak about evaluation cost in a semantics-driven way, we use a cost semantics: a big-step relation decorated with a natural-number cost, or an equivalent small-step machine with step-count. (This is standard in the cost-semantics and resource-analysis literature [DLR15].)

**Definition 2.15** (Costed evaluation)**.** A *costed operational semantics* for $L_{\mathsf{core}}$ is a relation

$$Q \Downarrow^{c_{\mathsf{eval}}} v$$

where $Q \in \mathrm{Prog}(L_{\mathsf{core}})$, $v$ is a value (or outcome), and $c_{\mathsf{eval}} \in \mathbb{N}$ measures evaluation cost (e.g. number of abstract-machine steps).

**Verification cost.** Verifiability (previous section) says that certain semantic claims can be certified by checkable artifacts. Tractability asks: are those artifacts and their checks *feasible*?

**Definition 2.16** (Certificates and checking cost)**.** Fix a specification class $\Phi$ (often induced by a type system) and a notion of certificate $\pi$. A *certificate checker* is a total procedure

$$\mathrm{Check}(P, \varphi, \pi) \in \{\mathsf{yes}, \mathsf{no}\}$$

such that $\mathrm{Check}(P, \varphi, \pi) = \mathsf{yes}$ implies $P \models \varphi$ (soundness). We write $c_{\mathsf{chk}}(P, \varphi, \pi)$ for the cost of running Check and $|\pi|$ for the size of the certificate.

This schema covers:

- ordinary typechecking (certificate implicit; $\pi$ is a derivation tree),
- proof-carrying code and certified resource bounds [CW00],
- proof terms in dependent type theories (certificate explicit; checking is typically decidable, but proof construction may be expensive) [CH88].

**The EVT notion: polynomial overhead, not polynomial runtime.** A key design choice: we do *not* define tractability as "all programs run in polynomial time." That would collapse tractability into ICC (and would make most general-purpose languages "intractable" by definition). Instead, we measure whether the *language-induced overhead* is polynomially bounded as a function of program size.

**Definition 2.17** (Pipeline cost)**.** Given an elaboration system and costed evaluation as above, define the total pipeline cost for a terminating $P$ by

$$c_{\mathsf{pipe}}(P) \;=\; c_{\mathsf{elab}}(P) \;+\; c_{\mathsf{eval}}(\mathrm{Elab}(P)).$$

If verification is included for a target specification $\varphi$ with certificate $\pi$, define the verified pipeline cost by

$$c^{\mathsf{ver}}_{\mathsf{pipe}}(P, \varphi, \pi) \;=\; c_{\mathsf{elab}}(P) \;+\; c_{\mathsf{chk}}(P, \varphi, \pi) \;+\; c_{\mathsf{eval}}(\mathrm{Elab}(P)),$$

and the verified artifact size by $|P| + |\pi| + |\mathrm{Elab}(P)|$.

**Definition 2.18** (Polynomial overhead tractability)**.** A surface language $L_{\mathsf{surf}}$ (relative to a core $L_{\mathsf{core}}$ and a fixed pipeline) is *tractable* if there exists a polynomial $p(\cdot)$ such that for every surface program $P$:

(T1) **Elaboration boundedness.** If $P \Downarrow^{c_{\mathsf{elab}}} Q$ then $c_{\mathsf{elab}} \le p(|P|)$ and $|Q| \le p(|P|)$.

(T2) **Checking boundedness (when applicable).** For the chosen verifiability discipline, whenever $\mathrm{Check}(P, \varphi, \pi) = \mathsf{yes}$ we have $c_{\mathsf{chk}}(P, \varphi, \pi) \le p(|P| + |\pi|)$ and (optionally, as a design desideratum) $|\pi| \le p(|P|)$.

Clause (T1) formalizes the intuition that "burying semantics in macros" can break tractability: if the language admits short programs whose elaborations blow up super-polynomially, then the core semantics may be simple while the surface language is not practically usable as a *compositional* notation.

**Relation to resource analysis.** There is an active PL literature on deriving machine-checkable resource bounds from high-level programs via type-based methods (automatic amortized resource analysis) and cost semantics [HH10, HAH12, DLR15]. These works are not "tractability of languages" in our EVT sense, but they provide exactly the semantic technology we need: explicit cost models, compositional accounting, and checkable certificates in some regimes [CW00]. The polynomial-overhead definition captures machine-level feasibility of the pipeline. But the EVT also wants to say something about *human* feasibility: whether programs and proofs are controllable without global mental backtracking. A useful bridge is to measure the structural complexity of the dependency graphs induced by a program or proof. Bounded-width graphs support efficient analyses (via dynamic programming), and they also correspond to lower "working set" requirements for reasoning.

**Definition 2.19** (Cognitive control width (CFG pathwidth proxy))**.** Let $\mathrm{CFG}(P)$ be a control-flow graph for $P$ (or for $\mathrm{Elab}(P)$) in a standard sense. Define the *cognitive control width* by

$$\mathrm{CCW}(P) \;=\; \mathrm{pw}(\mathrm{CFG}(P)),$$

where $\mathrm{pw}(\cdot)$ is graph pathwidth. Optionally, for proof-carrying regimes, one can analogously define $\mathrm{CCW}_{\mathsf{proof}}(P, \pi)$ using a proof-dependency graph built from the derivation structure.

Two concrete calibration facts (for context, not as axioms of the EVT):

- Structured (goto-free) programs have strong sparsity properties in their control-flow graphs; classic work shows bounded treewidth for broad classes of structured programs [Tho98].
- More recently, control-flow graphs of structured programs have been shown to have bounded *pathwidth* as a function of nesting depth, together with efficient construction of decompositions [CGL23].

These results justify using width as a tractability proxy: when the control (or proof) structure has bounded width, many analyses and transformations become feasible in practice, and the amount of "global context" required to reason locally is reduced.

**Human-side motivation.** On the HCI side, Green and Petre's *Cognitive Dimensions of Notations* provides a vocabulary for discussing human tractability (viscosity, visibility, error-proneness, etc.) and explicitly frames usability as a space of tradeoffs rather than a single scalar [GP96]. Width parameters are not a replacement for cognitive dimensions; rather, they are one formal knob that can partially explain why some notations and control structures are easier to manipulate and analyze than others.

**Conceptual compressibility.** Let $\approx_L$ be contextual equivalence (as in definition 2.2). Define the *behavioral Kolmogorov complexity* of $P$ by

$$K_{\approx}(P) \;=\; \min\{\, |Q| \;:\; Q \approx_L P \,\}.$$

Then the ratio $K_{\approx}(P)/|P|$ informally measures redundancy/ceremony. We will not use $K_{\approx}$ in formal theorems (it is non-computable), but it motivates the claim that "tractability" includes a human-facing component: how directly code reflects the essential structure of its behavior.

### 2.5 Macro-expression and Rice-genericity

This first clause should be read as a spectrum claim rather than as a single undecidability theorem. The EVT Hypothesis does not assert that every increase in expressiveness immediately destroys verification, nor that Rice's theorem is the universal source of all computational opacity. Rather, it asserts that once a computational substrate becomes sufficiently rich, global verification stops being available "for free." To recover it, one must impose structure: restrict the language, restrict the dynamics, restrict the observable class, provide certificates, or accept additional proof/compilation/human burden.

**Expressiveness and global verifiability.** Let $E$ denote the expressive capacity of a computational system: the range of behaviors, transformations, or observables it can directly realize. Let $V_{\mathrm{glob}}$ denote the range of global semantic properties of that system that admit uniform algorithmic verification. The first half of the EVT Hypothesis may then be read informally as the monotonic pressure

$$\Delta E > 0 \quad\Longrightarrow\quad \Delta V_{\mathrm{glob}} < 0 \;\text{ unless compensating structure is introduced.}$$

The word "global" is essential. Local verification may remain robust: type safety, conservation laws, local rewrite invariants, stabilizer fragments, syntactic safety properties, and bounded model checks can all remain available in expressive systems. The claim concerns broad semantic properties whose verification requires understanding the system as a whole.

**Classical symbolic endpoint: Rice-style opacity.** In deterministic symbolic computation, the limiting case is familiar. Once a language admits universal evaluation, effective specialization, and extensional semantic classification, Rice's theorem applies: every nontrivial extensional property of the partial function computed by a program is undecidable [Ric53, Rog67, Soa87]. In this regime, macro-expressive features such as reflection, quotation/evaluation, interpreter embedding, and rich metaprogramming tend to supply the structural conditions under which self-reference and diagonalization become routine. Verifiability is then recovered only by withdrawing some of that fuel: totality, stratification, restricted reflection, syntactic safety disciplines, or proof-carrying certificates.

**Quantum/physical endpoint: noncommutative opacity.** Physical computation illustrates a different endpoint. A von Neumann machine can simulate complex-linear or noncommutative operations in software, but its native execution model is still a deterministic update over a fixed classical state representation. Quantum computation, by contrast, has a native Hilbert-space semantics: gates are unitary transformations and observable quantities need not commute [NC10]. The unitary evolution of a closed quantum circuit is deterministic, but measurement produces probabilistic outcomes via the Born rule. More importantly for EVT, incompatible observables do not admit a single global classical valuation; contextuality and noncommutativity obstruct the reduction of all behavior to one globally inspectable semantic summary [KS67, AB11]. Here the loss of global verifiability is not primarily Rice-style self-reference, but native noncommuting composition and measurement-context dependence.

**The common pattern.** Rice and quantum contextuality therefore play different roles. Rice is the symbolic/deterministic endpoint of global semantic opacity: the obstruction arises from program space, self-interpretation, and extensional classification. Quantum computation is a physical/noncommutative endpoint: the obstruction arises from the native structure of observables and measurement. EVT treats both as instances of a broader pattern:

$$\text{rich composition} \implies \text{loss of global transparent semantics} \implies \text{verification requires additional structure.}$$

In the symbolic case that structure may be typing, totality, stratification, or certificates. In the quantum case it may be restriction to commuting observables, stabilizer/Clifford fragments, restricted measurement classes, error-correcting structure, or classically simulable subtheories.

**Relation to the Second Law.** This formulation aligns the first half of the EVT Hypothesis with the Second Law of Programming Thermodynamics. The Second Law states that, above a threshold of semantic richness, increasing verifiability generally requires an entropic loss elsewhere: reduced expressiveness, increased proof burden, stronger automation/oracles, or degraded compilability/comprehensibility. The present clause identifies the same phenomenon from the opposite direction: increasing expressiveness generally shrinks the class of globally verifiable behaviors unless compensating structure is added. The "tradeoff" is therefore not a literal conservation equation, but a directional pressure on computational design.

## 3 EVT: Scenarios and Failures

We now turn to the second half of the EVT: not the definitions of the axes, but the ways systems of computation begin to fail when pushed toward the boundary. Our strategy is to study *failure modes* obtained by holding two constraints fixed (or "high") while attempting to maximize the remaining dimensions.

Throughout this section, keep in mind the intended meaning of our guardrails:

- **Consistency.** Accepted proofs/certificates correspond to true semantic claims.
- **Compilability.** The elaboration–checking–evaluation pipeline has controlled (e.g. polynomial) overhead, as in section 2.4.
- **Comprehensibility.** The system remains usable as a human interface, rather than an opaque proof-search engine.

We treat comprehensibility as a distinct (human-facing) constraint even though it partially overlaps with our broader tractability discussion.

### 3.1 Gödel's Type System

**Loss of Consistency.** A *Gödel type system* (in our sense) is a type system in which *the proof language is computationally universal* in a way that meaningfully interacts with typing/definitional equality. The name is meant to evoke Gödelian diagonalization: once sufficiently strong self-reference and computation are internalized, some combination of decidability, completeness, and consistency must fail.

**Definition 3.1** (Gödel Type System)**.** Let $\mathcal{T}$ be a type system with a distinguished empty/false type $\bot$ ("False") and the Curry–Howard reading (types as propositions, terms as proofs) [How80]. We call $\mathcal{T}$ a *Gödel type system* if there is a computable map

$$\langle M, x\rangle \longmapsto \tau_{M,x}$$

from Turing machines $M$ and inputs $x$ to *closed* types $\tau_{M,x}$ such that

$$\tau_{M,x} \text{ is inhabited in } \mathcal{T} \quad \Longleftrightarrow \quad M \text{ halts on input } x.$$

Equivalently, the inhabitation problem for $\mathcal{T}$ is at least as hard as the halting problem.

Even relatively disciplined polymorphic calculi already exhibit undecidable *typability/type-checking* problems in Curry-style formulations [Wel99, Wel94]. Gödel type systems are the "maximal" version of this phenomenon: once inhabitation can encode halting, no sound and complete mechanical verifier for inhabitation can exist. This failure mode occurs when one maximizes comprehensibility and compilability while still keeping a high level of expressiveness ($E$). This triggers an *unsound* failure mode.

This is the realm of a language where one can express quickly, compile fast, and understand easily. Only to be met with the situation where they must then remark

> *"...hold a second, why did my proof of* $1 = 0$ *pass typechecking?"*

The danger zone is precisely where the language is simultaneously expressive enough that nontrivial computation can be encoded *at the proof/type level*, and engineered for fast, predictable checking, with a simple mental model. Under these pressures, systems frequently "solve" the conflict by silently weakening the meaning of "proof accepted"—i.e. by becoming unsound.

**Core mechanism of failure.** At a high level, the pipeline is:

(high $E$ at the type level) $\Longrightarrow$ (general computation inside proofs) $\Longrightarrow$ (loss of normalization/decidability).

At that point, there are only a few options:

- (U1) sacrifice tractability (permit expensive/global proof search or heavy automation),
- (U2) sacrifice expressiveness (restrict recursion/effects; stratify universes),
- (U3) or sacrifice soundness (accept an incomplete/approximate kernel that may accept bogus inhabitants of $\bot$).

The unsound failure mode is (U3).

A canonical way soundness collapses is by admitting *general recursion as proof*.

**Proposition 3.1** (General recursion trivializes the logic)**.** *Let $\mathcal{T}$ be a Curry–Howard style type system with:*

- *(H1) an empty type $\bot$ and an elimination principle $\mathsf{exfalso}_A : \bot \to A$ for every type $A$ (principle of explosion), and*
- *(H2) a fixed-point operator $\mathsf{fix}_A : (A \to A) \to A$ for every type $A$ (general recursion at the term/proof level).*

*Then every type $A$ is inhabited (the logic is inconsistent).*

*Proof.* Let $f \triangleq (\lambda x{:}\bot.\ x) : \bot \to \bot$. Define

$$\mathsf{loop} \triangleq \mathsf{fix}_\bot(f) : \bot.$$

Then for any $A$, define $\mathsf{explode}_A \triangleq \mathsf{exfalso}_A(\mathsf{loop}) : A$. □

**Coq vs. Idris on type safety.** Proof assistants based on normalization (e.g. Coq's CIC) enforce termination/guardedness conditions precisely to prevent theorem 3.1. Coq's kernel uses a syntactic guardedness/structural recursion criterion (the "guard condition") rather than admitting unrestricted fixed points [Gim95, Bar06, CH88]. But what if we allow general recursion at the proof level? Imagine we write something like this in a dependently typed language *without* a termination/totality discipline enforced by the kernel (or with it disabled) [Bra13]:

```
loop : False
loop = loop
```

This defines a "proof" of `False` that is just itself. Once we have `loop : False`, we can write:

```
explode : a
explode = case loop of {}
```

This is the principle of explosion: from $\bot$ one can produce anything. In other words, once $\bot$ is inhabited, *every* proposition becomes provable, and the proof system collapses.

**Formal version: the Curry paradox.** The above is one route (general recursion). A second, more "Gödelian" route is *self-reference*. Curry's paradox shows that if a system admits sufficiently strong self-referential propositions of the form

$$C \;\leftrightarrow\; (C \to F),$$

then it can derive the arbitrary conclusion $F$ (hence prove everything) [Cur42, Sha17]. One clean way to package this for the EVT is as a schema.

**Proposition 3.2** (Curry fixed-point schema implies triviality)**.** *Work in a logic/type theory with implication and contraction. Assume the following* fixed-point schema*: for every formula/type-valued expression $\varphi(X)$ (with $X$ occurring only positively), there exists a closed formula/type $C$ such that*

$$C \;\leftrightarrow\; \varphi(C).$$

*Then for every formula $F$, the logic proves $F$.*

*Proof sketch.* Take $\varphi(X) \triangleq (X \to F)$, so $C \leftrightarrow (C \to F)$. From $C$ and $C \to F$, we get $F$ by modus ponens. But $C$ also implies $C \to F$, so one can derive $C$ and hence $F$; Curry's original presentations isolate exactly which structural rules are responsible (notably contraction). □

**Type-theoretic analogues.** Type theories encounter Curry-like self-reference pressures whenever they blur critical stratifications: e.g., collapsing universe levels ("Type : Type") yields Girard/Hurkens-style paradoxes that derive $\bot$ [Coq86, Hur95]. Modern proof assistants, therefore, pay an explicit complexity tax (universe hierarchies, guarded recursion, syntactic restrictions) to remain sound. This failure mode is the cautionary tale behind the First Law. If one wants:

- a language that is easy to write and understand (high comprehension),
- a compiler/checker that runs fast and predictably (high compilability),
- and also type-level expressiveness that can internalize serious computation (high $E$),

then they are constantly tempted toward Gödelian constructions: either general recursion sneaks into proofs, or self-reference/impredicativity collapses the stratification that keeps the logic consistent. Unless one imposes strong guardrails (totality/termination, universe stratification, restricted reflection), the system becomes unsound.

*In summary.* The type system eats the language.

### 3.2 Tarski's Compiler

**Loss of Compilability.** A *Tarski compiler* is a compiler (more broadly: an elaborator/typechecker pipeline) that is asked to produce *truth inside itself*.

The name is deliberate. Tarski's undefinability phenomenon can be summarized informally as: once a system is expressive enough to encode arithmetic, it cannot contain a truth predicate for its own sentences satisfying the full "$T$-schema" without collapse [Tar36]. We use this as an analogy—not a literal reduction—for what happens when a language design tries to make the compiler silently discharge arbitrary semantic obligations while remaining both correct and human-friendly.

**Compilability as a working constraint.** In the scenario analysis, *compilability* is the requirement that the compilation pipeline is a *total, predictable procedure*: given a program, it should either (a) produce compiled output, or (b) reject with a diagnostic, within controlled resource bounds. When compilation includes elaboration, implicit argument inference, unification, and proof search, compilability becomes a substantive constraint rather than a default expectation.

**Definition 3.2** (Tarski compiler)**.** Fix a language $L$ whose static discipline expresses a class of propositions/specifications $\Phi$ (often via types). A compiler pipeline $\mathsf{Comp}$ for $L$ is called a *Tarski compiler* if it is expected to satisfy all three of the following:

(TC1) **Consistency.** Whenever $\mathsf{Comp}$ accepts a program, the accepted implicit obligations are semantically valid (soundness of compilation and checking).

(TC2) **Comprehensibility.** Programmers may omit large parts of the proof/annotation burden; the system attempts to infer or synthesize missing logical structure "in the background."

(TC3) **Internal truth production.** For a wide class of implicit obligations $\varphi \in \Phi$, the compiler is expected to *decide* (or effectively settle) whether $\varphi$ holds, as part of determining whether the program compiles.

The tension is immediate: (TC3) is a demand that compilation act like a truth oracle for a rich internal logic, while (TC2) shifts responsibility from the human to the compiler. This is the second failure state, the *Tarski Compiler*. It occurs when one designs a language that is highly expressive ($E$) and insists on consistency (soundness/verifiability), but also tries to remain comprehensible—meaning that ordinary programmers can write code without supplying full formal proofs. The goal is to remain powerful and safe, but still human-friendly; then the price is that the compiler inherits the missing work: elaboration becomes proof search.

**Core mechanism of failure.** The design typically includes all of the following:

- a static logic expressive enough to encode nontrivial propositions (e.g. dependent types, rich refinement constraints, higher-order encodings);
- a correctness mandate: accepted programs must come with valid certificates (explicit or implicit);
- an ergonomic mandate: the user does not write the certificates in full, so the compiler tries to infer/synthesize them.

As a result, compilation begins to include:

- **Typechecking as theorem proving.** "Does this elaborate/typecheck?" becomes "is there a proof of this obligation?"
- **Inference as proof search.** The compiler runs search procedures (unification, instance resolution, tactic-like automation).
- **Compile-time nontermination/blowup.** These procedures are, in general, not guaranteed to terminate or to remain within predictable bounds.

The issue is that the compiler is forced into semi-decision procedures. The moment the compiler is responsible for *finding* proofs rather than merely *checking* them, it inherits the meta-theoretic limits of proof search. Even when the kernel checking problem is decidable (as it often is in normalization-based systems), the *synthesis* and *inference* problems that make the system comprehensible can be

undecidable or only semi-decidable. A canonical example is *higher-order unification*, which arises naturally during elaboration of dependently typed terms; it is not decidable in general and is typically handled by search/backtracking heuristics [Hue75, Hue76].

**Proposition 3.3.** *Assume a language $L$ whose specification logic $\Phi$ is expressive enough that proof existence (inhabitation) can encode an undecidable problem (e.g. the halting problem, as in definition 3.1). Consider a compiler pipeline that aims to satisfy:*

*(a) correctness (soundness: it never accepts false obligations), and*

*(b) comprehensibility (it attempts to infer/synthesize omitted proof obligations from a broad class).*

*Then, in general, the pipeline cannot satisfy compilability as a total decision procedure (always halts with accept/reject) for that broad class of programs/obligations.*

*Proof sketch.* If the compiler always halts and is correct, and if it is also sufficiently complete about inferring the omitted obligations (so that "easy to write" programs are accepted whenever the corresponding obligations are true), then it would decide truth/existence of proofs for a class of obligations that can encode an undecidable problem. This contradicts the assumed expressiveness of $\Phi$. Therefore, at least one of the desiderata must fail: either the compiler becomes incomplete (rejects/gets stuck on true programs), or it becomes nonterminating/unpredictable (loss of compilability), or the soundness/correctness guarantee is weakened. □

**Real-world examples.** This failure mode is the compiler being asked to do the following, on demand:

> *"Here is a vague, high-level, partially specified statement. Please reconstruct the missing formal proof that makes it valid, and do so while I wait for compilation."*

Sometimes the proof does not exist. Sometimes it exists but requires a search space so large that compilation becomes effectively unbounded. In the limit, compilation becomes an interactive theorem-proving session that has been hidden behind the `build` command. This phenomenon is visible whenever automation is made "too helpful" in expressive proof-oriented languages:

- In Coq, aggressive automation and search (e.g. `auto`, `eauto`, typeclass resolution) can lead to severe performance degradation or nontermination on poorly constrained goals; practitioners mitigate this with search bounds, hints databases discipline, and explicit proof structure.[2]
- In Agda (and related systems), elaboration and instance search similarly require careful scoping and user guidance to avoid runaway search.

*In summary.* The language eats the compiler.

### 3.3 Rice's Revenge

**Loss of Comprehensibility.** *Rice's Revenge* occurs when a language's sophistication exceeds its user base.

A natural objection to the EVT is: if a type system is powerful enough, why can we not have maximal expressiveness and maximal verifiability simultaneously? Wouldn't paradoxes (Gödel-style) or nontermination (Tarski-style) automatically prevent a system from becoming "too perfect"? Why introduce a third, seemingly opaque axis about human brains?

The missing point is that "verification" is not a free property one adds on top of a language. It is itself a *language of certificates*—typing derivations, coercions, proof terms, solver traces, or tactic scripts—together with a kernel that checks them. Once a system is sufficiently expressive, the certificate language becomes a programming language in its own right, with its own search problems, its own size blowups, and its own usability constraints.

In other words:

$$\text{(rich specs)} \quad \Rightarrow \quad \text{(rich proof objects)} \quad \Rightarrow \quad \text{(somebody must build them).}$$

[2] This is not a criticism of Coq: it is an illustration that *proof search* is a different computational regime than *proof checking*.

If the compiler builds them, we risk the *Tarski Compiler* (loss of compilability). If the compiler accepts them without adequate checking, we risk *Gödel's Type System* (loss of correctness). The remaining option is that the *programmer* builds them—and that is where comprehensibility becomes a hard constraint rather than a cosmetic one.

This is the third failure state. It arises when we insist on:

- **Consistency.** Accepted programs really satisfy their specifications (soundness), and
- **Compilability.** Compilation/checking remains a predictable, terminating procedure,

while also pushing expressiveness $E$ and verifiability $V$ very high. The only remaining place for the irreducible difficulty to go is the *user*.

**A minimal formal model: explicit certificates.** Fix a surface language $L_{\mathsf{surf}}$ and an explicitly typed/certified core $L_{\mathsf{core}}$. Think of $L_{\mathsf{core}}$ as the language in which all implicit proof obligations have become explicit terms (coercions, proof arguments, equality witnesses, termination metrics, etc.) This "explicit core" pattern is standard in the design of verified compilers and proof-oriented languages: it is how one keeps the kernel small and checking syntax-directed.

**Definition 3.3** (Proof-carrying elaboration interface)**.** A proof-carrying compilation interface consists of:

(PC1) a checkable judgment in the core (typechecking/certificate checking)

$$\vdash_{\mathsf{core}} Q : \tau$$

that is decidable (or at least engineered to be reliably terminating) for $Q \in L_{\mathsf{core}}$;

(PC2) an erasure/forgetful map $\mathsf{erase} : L_{\mathsf{core}} \to L_{\mathsf{surf}}$;

(PC3) an elaboration relation producing *some* certificate-bearing core term for a given surface term:

$$P \rightsquigarrow Q \quad \text{with} \quad \mathsf{erase}(Q) = P.$$

This interface makes a clean separation:

- *checking* is tractable/decidable (compilability preserved), and
- *finding* the missing certificates is an additional problem not guaranteed to be automatic.

(See, e.g., the explicit-core approach in dependent extensions of Haskell, where decidable, syntax-directed checking is recovered by carrying explicit equality/coercion evidence [WVAdAE17].)

**Rice's theorem pressure: why inference cannot save us in general.** Rice's theorem says that every nontrivial extensional semantic property of programs is undecidable. While Rice is classically stated for partial computable functions, the moral transfers directly: once a specification language is rich enough to express nontrivial behavioral claims, there is no general total procedure that can infer (or decide) those claims from arbitrary programs. This yields the "revenge":

> *If one wants consistency and compilability in a highly expressive verification regime, then one cannot hide the semantic difficulty in the compiler. The difficulty reappears as proof burden and annotation burden for the human.*

**Proposition 3.4.** *Assume a setting where the verified meaning of a program requires producing certificate artifacts (proofs, coercions, termination evidence, etc.) that are checked by a sound, terminating kernel. If the property class being enforced includes nontrivial semantic properties (in the sense of Rice), then there is no general algorithm that, for all programs, can automatically synthesize the required certificates while remaining both correct and terminating. Therefore, preserving consistency and compilability forces certificate construction to become (at least partially) an interactive, user-driven activity—which can destroy comprehensibility at scale.*

*Proof sketch.* If a compiler were to (a) always terminate, (b) be correct (sound), and (c) automatically synthesize certificates for a sufficiently rich class of semantic obligations, then it would decide a nontrivial semantic property class uniformly, contradicting Rice-style undecidability. Hence, at least one of (a)–(c) must be relaxed. In this failure state, we refuse to relax (a) and (b), so we necessarily relax (c) by shifting work to the programmer. □

**The core mechanism of the failure.** In a system that remains correct and compiler-friendly, the "missing work" manifests as explicit proof structure: termination arguments, invariants, equality rewrites, and transports. Even when each individual proof step is local, the aggregate burden can be overwhelming. The result is a system that is sound and predictable but only usable by a small population trained to manage proof objects and their algebra.

To illustrate the third failure state, let us consider the construction of a hypothetically *maximally verifiable* and *maximally expressive* programming language. Introducing MAXLANG™.

**Design slogan.** MAXLANG aims to be:

- **Maximally expressive.** Full general recursion, effects, metaprogramming, reflection, the works.
- **Maximally verifiable.** Every exported behavior must come with a machine-checkable certificate.
- **Compiler-friendly.** The kernel only *checks* certificates; it does not perform open-ended proof search.

In other words, MAXLANG preserves consistency and compilability by making the programmer supply whatever the logic requires. The result is predictable compilation—at the cost of comprehensibility.

**A small program in MAXlang.** Here is "hello world"—not in the sense of printing a string, but in the sense of performing a mundane task: sorting a list of integers.

```
module Demo.Sort

-- Built-ins (informal): Nat, Int, List, (<=), etc.
-- Dependent subset type {x : A | P x} is written:  Subset A P
-- Proof terms are explicit; rewriting requires explicit equality transport.
-- Termination is required for definitional equality; general recursion exists
-- but must carry an explicit semantic certificate when exported.

-- A predicate that a list is sorted.
Sorted : List Int -> Type
Sorted []            = Unit
Sorted [x]           = Unit
Sorted (x :: y :: t) = (x <= y) × Sorted (y :: t)

-- A predicate that two lists are permutations (multiset equality).
Perm : List Int -> List Int -> Type
Perm xs ys =  (CountEq xs ys)   -- assume standard multiset counting spec

-- The "specification type" for a sorting function:
-- It must return a list that is sorted AND is a permutation of the input.
SortSpec : Type
SortSpec = (xs : List Int) -> { ys : List Int | Sorted ys × Perm xs ys }

-- Now we write insertion sort, but MAXlang requires evidence at each step.
insert : (x : Int) -> (ys : List Int) ->
         { zs : List Int | Sorted ys -> Sorted zs × Perm (x :: ys) zs }
insert x [] =  [x]  with
  proof Sorted[]ToSorted[x]  : Sorted [] -> Sorted [x]
  proof PermInsertNil        : Perm (x :: []) [x]

insert x (y :: t) =
  if x <= y then
    (x :: y :: t) with
      proof SortedInsertFront :
        Sorted (y :: t) -> Sorted (x :: y :: t)
      proof PermInsertFront :
```

```
        Perm (x :: (y :: t)) (x :: y :: t)
    else
      let rec = insert x t in
      let zs  = rec.value in
      (y :: zs) with
        proof SortedInsertTail :
          (Sorted (y :: t)) ->
          (Sorted zs) -> (y <= head zs) ->
          Sorted (y :: zs)
        proof PermInsertTail :
          Perm (x :: (y :: t)) (y :: zs)

-- The actual sort function.
sort : SortSpec
sort []       = [] with proof TrivialSortedNil : Sorted []
                         proof TrivialPermNil     : Perm [] []
sort (x :: t) =
  let ih  = sort t in
  let ys  = ih.value in
  let prS = ih.proof.left  in     -- Sorted ys
  let prP = ih.proof.right in     -- Perm t ys
  let ins = insert x ys in
  ins.value with
    proof FinalSorted :
      -- combines prS with the insert proof:
      Sorted (ins.value)
    proof FinalPerm :
      -- combines prP with the insert proof:
      Perm (x :: t) (ins.value)
```

There is nothing exotic about sorting. The burdensome part is that MAXLANG forces every mundane function to carry:

- a *semantic specification* (here: sortedness and permutations),
- explicit proof terms (or proof skeletons) for every nontrivial step,
- explicit transports/rewrites (hidden above behind the word `proof`).

The kernel remains small, and compilation remains predictable—because the compiler is not searching for these proofs. But the *surface language* has become a proof engineering environment, even for tasks that normal programmers expect to be one screen of code.

*In summary.* The language eats the programmer.

Table 1: Failure Modes of the EVT Trilemma

| **Preserved** | **Broken Constraint** | **Resulting Pathology** | **Failure Mode** |
|---|---|---|---|
| Consistency Compilability | Comprehensibility | Programmers need to be PhDs in Category Theory and Modal Logic to use the language | Rice's Revenge |
| Consistency Comprehensibility | Compilability | A compiler or interpreter may never halt due to undecidable analyses or infinite proof search | Tarski Compiler |
| Compilability Comprehensibility | Consistency | Language is pleasant to use, but makes no guarantees about behavior or correctness | Godel Type System |

Hence, for any sufficiently expressive language, there exists a bound on the combined achievable levels of expressiveness, verifiability, and comprehensibility. Pushing all three to their extremes leads to logical inconsistency (breakdown of soundness), algorithmic intractability (undecidability or non-termination of analysis), or human incomprehensibility (cognitive overload).

### 3.4 Landin's Ghost?

**On hardware intolerance and expressivity.** Even if it could be designed, MAXLANG will likely never run on commodity hardware in the way ordinary "general-purpose" languages do. Why might this be the case?

Landin's remark about delivering LISP from "hand-to-mouth storage allocation" is a clue: the expressive affordances of a language are never purely syntactic. They come with a runtime model (allocation, representation, control, effects), and that runtime model has a *hardware footprint*. Historically, when the footprint was large enough, it motivated building machines around the language.

**Hardware intolerance.** We will call a language *hardware-tolerant* if it admits efficient implementations on a stable, commodity execution substrate (a fixed ISA and memory hierarchy), without requiring language-specific microarchitectural support beyond what is already ubiquitous. Conversely, a language is *hardware-intolerant* when its dominant costs are tied to semantic mechanisms that are poorly matched to the commodity substrate, so that acceptable performance requires either (a) extensive dynamic compilation machinery, or (b) specialized hardware support.

This is not meant as a sharp complexity-theoretic classification; it is an engineering distinction. Its purpose in the EVT is to explain why "maximal" expressiveness tends to drag the execution platform along with it.

**Why the Lisp Machine existed.** The MIT Lisp Machine project is the canonical example of hardware intolerance being treated as a first-class design constraint. The goal was not merely to run LISP *at all*, but to run it efficiently by moving key semantic mechanisms into the architecture: tagged representations (data with built-in type fields), microcoded support for high-level operations, and storage-management features such as garbage collection tightly integrated with the machine model [BMW77, Gre80, Moo84]. The design philosophy is explicit in the literature: Lisp machines are "high-level language computer architectures," where the boundary between "language runtime" and "hardware" is intentionally blurred [Ste02, WM81].

On that reading, Landin's warning is not aesthetic. It is architectural: certain forms of expressivity (especially those built around pervasive allocation, dynamic representation, and reflective control) create a pressure gradient toward bespoke execution support.

**Modern day situation.** But excluding computer history museums, retro-hacker conventions, or perhaps the basements of various DARPA/Air Force research labs, one does not find working Lisp machines.[3] Modern commodity computing largely converged on a small number of ISA families and broadly "RISC-like" microarchitectural principles, driven by VLSI design constraints, toolchain/ecosystem lock-in, and economies of scale [PD80]. Even when contemporary CPUs are not purely RISC in an ISA sense, their performance envelope is dominated by a standardized substrate: pipelines, caches, branch predictors, SIMD, and a memory hierarchy optimized for common workloads. In such an environment, languages that demand unusual runtime services must either simulate them in software (VMs, JITs, GC) or accept large constant factors.

**Kogge and the architecture of symbolic computers.** A complementary perspective comes from Peter Kogge's textbook *The Architecture of Symbolic Computers*, which treats "symbolic" execution models (functional and logic language computing, dynamic representations, heap-centric computation, graph reduction, pattern-driven control) as architectural first-class citizens rather than mere software conventions [Kog91]. Kogge's central theme aligns with the Lisp Machine lesson: once a language's semantic core relies heavily on mechanisms such as tagged representations, pervasive allocation, indirect addressing, and runtime-managed structure (including garbage collection), the performance-critical

[3] The rumor that Symbolics Lisp Machines were the main backbone behind the infamous Cold War era "Star Wars" program was, in all likelihood, an elaborate hoax perpetrated by early Usenet trolls.

path is no longer "a fast inner loop" on a stable instruction set, but a tight interplay between evaluation strategy, memory system behavior, and runtime services.

In such regimes, the naive von Neumann/RISC abstraction (a fixed ISA executing a uniform stream of instructions over unstructured memory) becomes a poor semantic match. Efficiency pressures naturally reintroduce language-specific support: either in the architecture (tags, specialized primitives, microcode-like support) or in an increasingly elaborate compilation/runtime tower (JITs, staged specialization, and aggressive representation management). This is precisely the sense in which highly expressive symbolic substrates can become *hardware-intolerant*: the more the language shifts computation into rich runtime structure, the more the "machine" must effectively internalize that structure to execute it efficiently.

**MAXlang pushes the pressure to the limit.** MAXLANG is intentionally designed to maximize expressiveness and verifiability. But note what that means operationally:

- *Expressiveness* pushes semantic phenomena into the language (reflection, metaprogramming, rich effects, intensional equality, etc.).
- *Verifiability* pushes obligations into the static world (proof terms, equality witnesses, termination/totality artifacts, refinement evidence).

The combined result is that a "program" is no longer merely an instruction stream. It is a coupled object: code *plus* certificates *plus* a meta-level that may compute with those certificates.

At that point, the classical separation between "compile" and "run" becomes unstable. One principled way to see this is through program specialization: compilation can be viewed as specializing an interpreter (or semantic evaluator) to a fixed source program (Futamura's perspective) [Fut99, JGS93]. When the semantic apparatus is simple, specialization yields a stable target (machine code for a fixed ISA). When the semantic apparatus includes rich metalevel computation and proof-relevant evaluation, efficient execution increasingly looks like *specializing the evaluator itself* to the particular program's proof obligations and reflective behavior. In the limit, *each program induces its own optimal abstract machine*.

This is the precise sense in which "every program defines its own hardware": not that literal silicon must be fabricated per program, but that a fixed universal substrate becomes a poor match. The natural endpoint is either:

- a tower of staging/JIT/specialization machinery that effectively synthesizes a program-specific execution strategy, or
- reconfigurable computing (microcode/FPGA-like specialization), where the "instruction set" is itself a residual artifact of compilation.

Either way, the notion of a single, uniform "universal processor" that runs MAXLANG efficiently for all programs becomes aspirational rather than realistic. Taken together, they explain why a universal "do-what-I-mean" compiler cannot exist: a compiler that is asked to be both reflective and semantically omniscient is being asked for a truth predicate over its own behavioral universe. On the physical side, there are genuine thermodynamic limits on information processing (e.g. Landauer's principle relating logical irreversibility to heat dissipation) [Lan61]. We invoke such results only as a reminder that computation is not free: unbounded elaboration, unbounded proof normalization, and unbounded meta-level execution are not merely theoretical inconveniences—they collide with finite time, energy, and memory.

**Landin's warning, updated.** Landin's gumblings about Lisp's appetites (allocation discipline, pedagogical heaviness, syntactic and semantic overhead) read today like a warning about the whole "maximal language" fantasy [Lan66]. In the MAXLANG limit, the appetite is no longer just for heap cells; it is for compilation effort, proof artifacts, and meta-level computation. And in an era of LLM-aided programming, where abstraction layers can be generated faster than they can be understood, the pressure toward hardware-intolerant semantics is stronger rather than weaker.

## 4 Modeling Expressiveness vs. Verifiability Tradeoffs

We now sketch a formal model for an expressiveness/verifiability (E/V) frontier. For the moment, we set aside the third axis (tractability in the broad sense, and the three-$C$ failure analysis in section 3) and work purely in two dimensions. The correct mathematical setting for such a discussion is therefore not $\mathbb{R}^3$ but a *2D partially ordered* space of language designs.

### 4.1 Computation of E and V

It is tempting to define numerical functionals $E(L)$ and $V(L)$ directly. However, in any sufficiently rich universe of languages, both quantities are generically *non-computable* in the sense that deciding them exactly would require deciding nontrivial semantic properties.

**Expressiveness.** Determining whether one language can simulate another, or whether a given extension is macro-expressive, ultimately depends on the semantic equivalence/translation facts of the form

$$\exists\ \phi : L' \to L \text{ such that } \forall P,\ \mathrm{Obs}_{L'}(P) = \mathrm{Obs}_L(\phi(P)),$$

and these are, in general, not decidable uniformly. At the extreme end, asking whether a language can express all partial computable functions ("is it Turing complete?") is itself a semantic property, and many semantic properties become undecidable once a system is universal [Chu36, Tur37].

**Verifiability.** Likewise, "can we statically prove property $\varphi$ for arbitrary programs?" quickly collapses to halting- and Rice-style impossibility: if a static discipline were both sound and complete for a nontrivial extensional property class over a universal language, it would decide an undecidable problem.

We do not need closed-form computation of $E(L)$ or $V(L)$ in order to state and analyze boundary phenomena. The right move is order-theoretic: define robust *comparisons* ($\leq$) and then reason about the shape of the feasible region.

### 4.2 The E- and V-orders

Fix a universe $\mathcal{L}$ of languages (as in section 2.2), and restrict attention to conservative sublanguages/variants of $\mathcal{L}$.

**The expressiveness order.** Recall $\preceq_{\text{macro}}$ from definition 2.8: it compares languages by which features they can express via structure-preserving elimination. We will write

$$L_0 \preceq_E L_1 \quad \text{for} \quad L_0 \preceq_{\text{macro}} L_1,$$

and interpret this as: $L_1$ *is at least as expressive as* $L_0$ *(in the macro-expressiveness sense).*

**The verifiability order.** We similarly compare languages by the strength of their checkable static disciplines. There are many reasonable ways to define this (depending on what one takes as the specification class), but we only need a monotone comparison.

**Definition 4.1** (Verifiability preorder)**.** Let $L_0$ and $L_1$ be languages equipped with checkable verification disciplines $\mathcal{T}_0$ and $\mathcal{T}_1$ (type systems/certificate systems) in the sense of Definition 2.12. Write

$$(L_0, \mathcal{T}_0) \ \preceq_V \ (L_1, \mathcal{T}_1)$$

if every semantic guarantee certified by $\mathcal{T}_0$ (under the intended interpretation) can also be certified by $\mathcal{T}_1$ with at most polynomial blowup in certificate size and checking time.

This definition is intentionally schematic: it formalizes the idea that one verifier subsumes another as a proof discipline. It avoids requiring *completeness* (which is too strong in expressive settings) and focuses instead on what can be *certified soundly and checkably.*

**Numerical scales as presentations.** If we draw an "E/V curve" in $[0,1]^2$, we should treat $E(L)$ and $V(L)$ as *presentations* of these preorders by monotone maps into the unit interval:

$$E : (\text{Lang}, \preceq_E) \to ([0,1], \leq), \qquad V : (\text{Lang}, \preceq_V) \to ([0,1], \leq),$$

chosen for visualization rather than computability. Nothing in the core argument requires $E$ and $V$ to be computable functions.

### 4.3 Endpoints: Lambda-calculus as Calibration

To anchor the discussion, we fix two well-studied endpoints: the untyped $\lambda$-calculus (ULC) and the simply typed $\lambda$-calculus (STLC). These are not claims about *all* programming languages; they are calibration points inside the standard semantic universe used throughout PL theory [Pie02].

**Endpoint 1: Untyped lambda calculus (ULC).** The untyped $\lambda$-calculus is the archetypal "pure expression machine": it has only abstraction and application. In its standard form, it is computationally universal: it can represent all partial computable functions (Church/Turing thesis as instantiated by the Church encoding) [Chu36, Tur37, Bar84].

Concretely, ULC admits nontermination ubiquitously, e.g.

$$\Omega \triangleq (\lambda x.\ x\,x)\ (\lambda x.\ x\,x),$$

and no static structure in the calculus itself prevents such terms.

As an EV endpoint, ULC motivates the "high $E$, low $V$" corner:

- **High expressiveness.** Universality and minimal primitives make it a convenient target for embeddings.
- **Low verifiability.** Without a static discipline, essentially no nontrivial semantic guarantees are derivable compositionally from the syntax alone.

In the informal $[0,1]$ picture, ULC is the prototypical point $(E, V) \approx (1, 0)$.

**Endpoint 2: Simply typed lambda calculus (STLC).** The simply typed $\lambda$-calculus adds a static discipline (types and typing rules) that enforces a strong semantic invariant: *well-typed programs do not go wrong*. More strongly, STLC is *strongly normalizing*: every well-typed term terminates [Tai67, Gir72, Pie02].

This moves STLC toward the "high $V$" corner:

- **High verifiability.** Typechecking is decidable and supports standard progress/preservation soundness theorems [Pie02, WF94].
- **Lower expressiveness.** The calculus is not computationally universal (no general recursion without extensions), so it cannot express all partial computable functions.

In the informal $[0,1]$ picture, STLC sits near $(E, V) \approx (0, 1)$ ("low $E$, high $V$").

ULC and STLC demonstrate that even within the smallest plausible semantic universe, one already sees a sharp tradeoff. The goal of the EVT program is to generalize this observation from these toy calculi to the richer design space of modern languages and verification systems, and to identify the structural constraints that force the boundary.

### 4.4 The E/V Frontier as a Pareto Boundary in a Poset

The discussion above treats "the $E/V$ curve" as a visual guide. Formally, however, we will *not* attempt to define computable real-valued scores $E(L), V(L) \in [0,1]$ for arbitrary languages. Instead, we model tradeoffs order-theoretically: languages are compared by preorders, and the "boundary" is a set of Pareto-maximal designs.

**Language designs.** A *language design* in this section is a pair

$$\mathbf{L} \;=\; (L, \mathcal{T}),$$

where $L$ is a programming language (with an observation model) and $\mathcal{T}$ is a checkable verification discipline for $L$ (type system/certificate system) as in Definition 2.12.

**Expressiveness preorder.** We write

$$\mathbf{L}_0 \preceq_E \mathbf{L}_1 \quad :\Longleftrightarrow \quad L_0 \preceq_{\text{macro}} L_1$$

using the macro-expressiveness preorder from Definition 2.8. Intuitively: $\mathbf{L}_1$ can express (via structure-preserving eliminations) at least the abstraction mechanisms available in $\mathbf{L}_0$.

**Verifiability preorder.** We write

$$\mathbf{L}_0 \preceq_V \mathbf{L}_1$$

if every semantic guarantee that can be certified in $\mathbf{L}_0$'s discipline can also be certified in $\mathbf{L}_1$'s discipline with controlled overhead. One concrete working choice (sufficient for the EVT) is:

**Definition 4.2** (Verifiability subsumption (working))**.** Let $\mathbf{L}_i = (L_i, \mathcal{T}_i)$ for $i \in \{0, 1\}$. Write $\mathbf{L}_0 \preceq_V \mathbf{L}_1$ if there exists a polynomial $p(\cdot)$ such that: for every closed program $P$ of $L_0$, every specification $\varphi$ expressible/certifiable under $\mathcal{T}_0$, and every certificate $\pi$ with

$$\text{Check}_{\mathcal{T}_0}(P, \varphi, \pi) = \mathsf{yes},$$

there exist a corresponding program $P'$ of $L_1$, specification $\varphi'$ under $\mathcal{T}_1$, and certificate $\pi'$ such that:

$$\text{Obs}_{L_0}(P) = \text{Obs}_{L_1}(P'), \qquad \text{Check}_{\mathcal{T}_1}(P', \varphi', \pi') = \mathsf{yes},$$

and the overhead bounds hold:

$$|P'| + |\pi'| \;\le\; p(|P| + |\pi|), \qquad c_{\mathsf{chk}}(P', \varphi', \pi') \;\le\; p(|P| + |\pi|).$$

Intuitively: $\mathbf{L}_1$ can certify everything $\mathbf{L}_0$ can (up to reasonable encoding overhead), under a sound checker.

**The product preorder and the frontier.** We now combine the two axes into a single preorder.

**Definition 4.3** ($E/V$ product preorder)**.** Define

$$\mathbf{L}_0 \preceq_{EV} \mathbf{L}_1 \quad :\Longleftrightarrow \quad \mathbf{L}_0 \preceq_E \mathbf{L}_1 \;\text{ and }\; \mathbf{L}_0 \preceq_V \mathbf{L}_1.$$

Write $\mathbf{L}_0 \prec_{EV} \mathbf{L}_1$ if $\mathbf{L}_0 \preceq_{EV} \mathbf{L}_1$ and not $\mathbf{L}_1 \preceq_{EV} \mathbf{L}_0$.

**Definition 4.4** (Pareto-maximal $E/V$ designs)**.** Let $\mathcal{S}$ be a set of language designs (e.g. a family generated from a fixed core by adding features). The $E/V$ *frontier* of $\mathcal{S}$ is the set

$$\text{Frontier}_{EV}(\mathcal{S}) \;=\; \big\{\mathbf{L} \in \mathcal{S} \;:\; \nexists\, \mathbf{L}' \in \mathcal{S} \text{ with } \mathbf{L} \prec_{EV} \mathbf{L}'\big\}.$$

**Proposition 4.1.** *If $\mathbf{L}, \mathbf{L}' \in \text{Frontier}_{EV}(\mathcal{S})$ are distinct, then neither $\mathbf{L} \prec_{EV} \mathbf{L}'$ nor $\mathbf{L}' \prec_{EV} \mathbf{L}$ holds. Equivalently, the strict frontier elements are pairwise incomparable under $\preceq_{EV}$.*

*Proof.* If $\mathbf{L} \prec_{EV} \mathbf{L}'$ then $\mathbf{L}$ is not Pareto-maximal by definition of the frontier, a contradiction. □

**How the "curve" picture survives.** The informal picture of an $E/V$ boundary is best understood as a *visualization* of $\text{Frontier}_{EV}(\mathcal{S})$ under some chosen monotone coordinates (a "chart"). That is, one may map designs to $(x, y) \in [0, 1]^2$ by any monotone summaries of $\preceq_E$ and $\preceq_V$ for the sake of exposition. But the invariant object is the frontier in the preorder, not a literal function $V = F(E)$.

With this in place, ULC and STLC serve as calibration points (extreme corners) inside a chosen family $\mathcal{S}$, as discussed in Section 4.3.

### 4.5 A Programming Language Design Space

The formal definition of $\text{Frontier}_{EV}(\mathcal{S})$ in Definition 4.4 is only meaningful once we fix a *design space* $\mathcal{S}$. If $\mathcal{S}$ were taken to be "all languages," the preorder becomes too unconstrained to support clean statements. As in physics, we therefore begin with a deliberately simplified model class: a finite family of languages generated from a common core by toggling a small set of features.

**Core calculus.** Fix a baseline core $L_0$ (for exposition, STLC with products and unit is sufficient; see Section 4.3). We will generate variants by adding features on top of $L_0$. This does not assert that all real languages are STLC extensions; it asserts only that STLC is a clean reference point in which $E$ and $V$ are well-understood.

**Feature set.** Let $\mathsf{F}$ be a finite set of feature symbols. For concreteness, we use the following six knobs, each of which is standard in the PL literature and has well-known effects on expressiveness and verification:

$$\begin{aligned}\mathsf{F} \;=\; \{\; &\mathsf{rec} && \text{(general recursion / partiality)},\\ &\mathsf{state} && \text{(mutable references / heap state)},\\ &\mathsf{ctrl} && \text{(first-class control / continuations)},\\ &\mathsf{poly} && \text{(parametric polymorphism)},\\ &\mathsf{dep} && \text{(dependent types / propositions-as-types)},\\ &\mathsf{refl} && \text{(reflection / staging / metaprogramming)}\;\}.\end{aligned}$$

**Feature-generated languages.** For each subset $X \subseteq \mathsf{F}$, let $L_X$ denote "the language obtained by extending $L_0$ with exactly the features in $X$," together with a specified operational semantics and observation model. We do not need to define every $L_X$ in full detail here; it suffices that each feature corresponds to a conservative extension in the sense of Definition 2.1.

**Definition 4.5** (Feature lattice of language cores)**.** Let

$$\mathcal{L}_{\mathsf{F}} \;=\; \{\, L_X \mid X \subseteq \mathsf{F} \,\}.$$

Order $\mathcal{L}_{\mathsf{F}}$ by subset inclusion on features:

$$L_X \preceq_{\mathsf{feat}} L_Y \quad :\Longleftrightarrow \quad X \subseteq Y.$$

$(\mathcal{L}_{\mathsf{F}}, \preceq_{\mathsf{feat}})$ is a finite Boolean lattice. Joins and meets are given by union and intersection of feature sets. We use this only as a *combinatorial scaffold*: it provides a controlled space in which incomparability and multiple maximal points are unavoidable and visible.

**Verification disciplines as part of the design.** A language design in the EVT is not just $L_X$ but a pair $(L_X, \mathcal{T})$ where $\mathcal{T}$ is a checkable verification discipline for $L_X$ (Definition 2.12). The same core language may admit multiple verification regimes (ranging from simple type safety to rich dependent correctness). To keep the model lightweight, we parameterize verification by a small set of *regimes* rather than trying to model all proof assistants.

**Definition 4.6** (Verification regimes)**.** Let $\mathsf{R}$ be a finite set of verification regimes. For example:

- $\mathsf{R}_{\mathsf{ST}}$: simple safety typing (progress/preservation).
- $\mathsf{R}_{\mathsf{HM}}$: inference-oriented polymorphic typing (Hindley–Milner-style).
- $\mathsf{R}_{\mathsf{Dep}}$: dependent typing with explicit proofs (proof-carrying).
- $\mathsf{R}_{\mathsf{Auto}}$: dependent typing plus significant automation/inference/search.

For each $X \subseteq \mathsf{F}$ and each $r \in \mathsf{R}$, assume a corresponding discipline $\mathcal{T}_{X,r}$ when meaningful.

**Definition 4.7** (The controlled design space $\mathcal{S}$)**.** Define the design space

$$\mathcal{S} \;=\; \{\, (L_X, \mathcal{T}_{X,r}) \mid X \subseteq \mathsf{F},\; r \in \mathsf{R},\; \mathcal{T}_{X,r} \text{ is well-defined} \,\}.$$

We study $\text{Frontier}_{EV}(\mathcal{S})$ with respect to the preorders $\preceq_E$ and $\preceq_V$ from Section 4.4.

| Name (informal) | Features $X \subseteq \mathsf{F}$ | Verification regime |
|---|---|---|
| STLC (baseline) | $\emptyset$ | $\mathsf{R}_{\mathsf{ST}}$ |
| ML-like (practical FP) | $\{\mathsf{rec}, \mathsf{poly}, \mathsf{state}\}$ | $\mathsf{R}_{\mathsf{HM}}$ (plus safety typing) |
| Coq/Agda-like (proof-carrying) | $\{\mathsf{dep}\}$ (often w/ inductives) | $\mathsf{R}_{\mathsf{Dep}}$ (explicit proofs) |
| Coq+automation (search-heavy) | $\{\mathsf{dep}\}$ | $\mathsf{R}_{\mathsf{Auto}}$ |
| MAXlang™ (toy) | $\{\mathsf{rec}, \mathsf{state}, \mathsf{ctrl}, \mathsf{poly}, \mathsf{dep}, \mathsf{refl}\}$ | $\mathsf{R}_{\mathsf{Dep}}$ (no hidden search) |

Table 2: A few landmark points in the controlled design space $\mathcal{S}$. These are illustrative "coordinates" for the reader, not a classification of real languages.

**How the "curve" picture is recovered.** Within $\mathcal{S}$, the Pareto frontier $\mathrm{Frontier}_{EV}(\mathcal{S})$ is typically not a single-valued function $V = F(E)$ but a set of maximal points (an antichain; Theorem 4.1). Nevertheless, one can still draw an intuitive boundary in $(E, V)$-coordinates by:

- plotting any monotone summaries ("charts") of $\preceq_E$ and $\preceq_V$ on $[0,1]^2$, and showing the maximal points as an envelope; or
- restricting attention to a single *design trajectory* (a chain in the lattice) and plotting the induced tradeoff along that path.

This preserves the explanatory value of the ULC/STLC endpoints while keeping the formal object of study order-theoretic.

**Bridge to the three failure states.** The feature-lattice model also explains why the three scenario failures in Section 3 are not arbitrary stories. Pushing toward higher $E$ typically corresponds to adding features such as $\mathsf{rec}$, $\mathsf{state}$, $\mathsf{ctrl}$, and $\mathsf{refl}$. Pushing toward higher $V$ corresponds to strengthening regimes toward $\mathsf{R}_{\mathsf{Dep}}$ and $\mathsf{R}_{\mathsf{Auto}}$. The three failures then correspond to three ways of resolving the resulting pressure.

### 4.6 The Turing Tarpit and Real-life Programming Languages

The EVT Trilemma does not assert a conservation equality of the form "high $E$ implies low $V$" in all regimes. In particular, there exist computational regimes with *both* low expressiveness and low verifiability. Herein lies the infamous *Turing tarpit*: a setting in which "everything is possible" in principle (often even Turing completeness), but very little is *practically* expressible or maintainable.

The classic slogan is due to Perlis [Per82]:

> *"Beware of the Turing tarpit in which everything is possible but nothing of interest is easy."*

The point is that a language can be universal while still being "low $E$" in the macro-expressiveness sense: it may lack compositional abstraction mechanisms (modules, higher-level control constructs, data abstraction, hygienic macros, etc.), so that large programs devolve into ad hoc encodings. At the same time, it can be "low $V$" because it provides little static structure for ruling out errors or stating invariants.

**Examples.** Shell scripting languages (e.g. `sh`/`bash`) are expressive enough to orchestrate powerful computations via external processes, but their semantics are brittle (stringly-typed data, implicit coercions, dynamic failure modes) and they offer weak static guarantees. Esoteric languages such as Malbolge occupy the extreme corner: computation is possible, but the language is intentionally hostile to human reasoning and compositional development.

The tarpit clarifies what the EVT failure modes are *not*. Gödel/Tarski/Rice-style failures concern what happens when one pushes toward the upper-right boundary (high $E$ and high $V$) while trying to retain the relevant "three C's" constraints. Most real languages live far from that boundary: they trade away maximal verification, maximal expressiveness, or both, to remain usable and implementable. The tarpit is the opposite corner: not an advanced boundary failure, but a regime where both axes are weak and the "theoretical universality" is an unhelpful consolation.

The Turing tarpit reminds us that low-$E$ and low-$V$ regimes are common. The three boundary failure states (Gödel/Tarski/Rice) remind us that pushing "up and right" toward simultaneous high expressiveness and high verifiability encounters structural limits. The natural question is therefore: *where do most real languages sit in practice?*

**A caveat about "placing" languages.** We do not claim a canonical embedding of real languages into a numeric $[0,1]^2$ plane. Our formal object is the preorder frontier $\text{Frontier}_{EV}(\mathcal{S})$ (Section 4.4), and any picture is necessarily a *chart* (a monotone summary) plus a choice of which aspects of a language one is treating as the "language" (core calculus vs. full ecosystem, standard library, tooling, and idioms). What follows is, therefore, an intentionally informal calibration for the reader.

**High $E$, low $V$: dynamic and reflective families.** Languages in the Lisp/Scheme lineage are often experienced as "high expressiveness" due to macros, reflection, and flexible evaluation models. Their static verifiability is often weaker in everyday use, either because the core is dynamically typed or because much behavior is pushed into runtime conventions. This is not a defect; it is a deliberate choice that prioritizes rapid abstraction and metaprogramming over strong static guarantees.

**High $V$, lower $E$: inference-oriented typed families.** ML-style languages (and related inference-heavy typed traditions) often sit closer to the "high verifiability" corner: strong static typing, principled inference, and a well-developed metatheory yield robust safety guarantees with relatively low annotation burden. The corresponding cost is that certain expressive moves (unrestricted effects, reflection, highly dynamic patterns) are either discouraged, encoded indirectly, or require explicit machinery that changes the programming model.

**Balanced regimes: systems languages under pressure.** Many widely-used languages aim for a pragmatic middle: significant expressiveness, but with a strong enough static structure to make large systems reliable. Rust is an explicit attempt to push verifiability upward (memory safety and aliasing discipline) without sacrificing systems-level expressiveness. C++ sits differently: it is extremely expressive and performant, but its verifiability story often relies heavily on discipline, tooling, and conventions rather than uniform static guarantees. Both illustrate the central EVT point: "balance" typically means distributing burden across language design, compiler checks, libraries, and human process rather than maximizing a single axis in isolation.

**Why empirical placement is future work.** A serious, quantitative account of where languages lie would require choosing operational measures. Even within our controlled model $\mathcal{S}$ (Section 4.5), one could imagine multiple levels of sophistication:

- **Feature counts/capability profiles.** Treat $E$ as a monotone function of which macro-expressive facilities are present; treat $V$ as a monotone function of which classes of certificates are checkable under the intended discipline.
- **Human-facing measures.** Collaborate with HCI researchers to operationalize comprehensibility/interaction cost using established frameworks (e.g. cognitive dimensions) rather than anecdotes [GP96].
- **Cost/complexity measures.** Refine the notion of "verification/evaluation cost" using machine-independent complexity measures in the sense of Blum-style axiomatizations, allowing comparisons that do not depend on a particular machine model [Blu67].

We treat such measurement programs as important but orthogonal: the EVT thesis concerns the existence and character of the boundary, not a definitive cartography of all languages.

### 4.7 Attack Vectors Against the EVT

One might ask whether a clever adversary could "route around" the EVT guardrails by hiding expressiveness or proof burden behind indirection. The short answer is: they can *move* the difficulty around, but they cannot remove it. In practice, such attempts tend to collapse into one of the three failure states: loss of correctness, loss of compilability, or loss of comprehensibility.

**Threat model.** Imagine an adversary with excellent taste in type theory. Their goal is to build a language that *appears* to be simultaneously highly expressive, highly verifiable, and still friendly to humans and compilers (the three C's). What tricks might they try?

"'latex

(a) **Meta-languages and language embedding.**

**Attack.**
Keep a tiny surface syntax that looks harmless, while embedding a full proof language inside the meta-level through macros, reflection, staging, or quotation.

**EVT response.**
The proof/certificate workload still exists; it merely shifts phases. Either the compiler must normalize or check large meta-generated artifacts, producing loss of *compilability*, or the user must reason about the meta-language to debug failures, producing loss of *comprehensibility*.

**Heuristic detector.**
Elaboration or certificate-derivation size grows rapidly even when surface programs remain small.

(b) **Type-level computation disguised as "just types."**

**Attack.**
Allow type-level recursion, type families, implicit search, or dependent evaluation so that the type layer computes like the value layer, hiding substantial computation inside apparently simple signatures.

**EVT response.**
Either the checker becomes unpredictable, producing loss of *compilability*; the system imposes strict stratification or termination conditions, reducing $E$; or it risks Gödel-style unsoundness, producing loss of *correctness*.

**Heuristic detector.**
Type-dependency graphs and checking derivations exhibit high entanglement or width; local reasoning breaks down.

(c) **"Semantics without syntax": magical inference.**

**Attack.**
Remove annotations and ask the compiler to infer nearly everything—types, proofs, equalities, coercions—so that the user writes pleasant code while the toolchain silently performs deep theorem proving.

**EVT response.**
This is Section 3.2 in disguise. Either inference/search diverges or blows up, producing loss of *compilability*, or it becomes so opaque that fixing errors requires global expertise, producing loss of *comprehensibility*.

**Heuristic detector.**
Long inference chains and non-local error explanations correlate with large derivations and wide dependency structure, although opacity is not fully captured by the present metrics.

**Optional future metric: locality of repair.** One plausible refinement—if adversaries get too clever—is to formalize a notion of *annotation transparency* or *locality of repair*:

> A design is $k$-*local* if, whenever type inference fails on a program $P$, there exists a correction that can be derived (or diagnosed) using only $O(k)$ lines of local context.

We do not treat this as a new EVT axis here: it is difficult to formalize cleanly, and may be partially emergent from the structural-width proxies already discussed. But empirically, inference opacity is a primary failure mode in advanced typed systems: users do not quit because the system is unsound; they quit because they cannot *repair* it.

*Takeaway.* "Cheating" strategies mostly change *where* the cost appears: in the compiler, in the proof artifacts, or in the user's head. The EVT is precisely the claim that, beyond a certain expressive/verifiable regime, we cannot avoid paying it somewhere.

## 5 Toward a Computational Thermodynamics

As established above, total/strongly normalizing calculi gain verification power at the price of general recursion, while reflection and metaprogramming gain flexibility at the cost of global guarantees. The natural question is whether other *computational formalisms* exhibit analogous tradeoffs once they are pushed toward physical or mathematical fidelity.

We emphasize that any such analogy is *discipline-dependent*: what counts as "expressiveness," "verifiability," and "tractability" depends on the computational regime under study. The EVT thesis here is not a literal transfer of PL theorems to physics, but a claim that similar boundary phenomena recur when one demands (a) a rich model class, (b) controlled guarantees, and (c) feasible computation.

### 5.1 Lattice Gauge Theory

Lattice gauge theory provides a particularly clear case study. It is a computational formalism for nonperturbative quantum field theory: one discretizes spacetime, integrates over large configuration spaces, and attempts to recover continuum observables with controlled systematic errors.

**A computational reading.** In an EVT-like translation:

- **Expressiveness** corresponds to the richness of the modeled dynamics: gauge fields with (in the continuum) infinitely many degrees of freedom, strong coupling phenomena, and path-integral expectation values over large configuration spaces.
- **Verifiability** corresponds to the ability to make *controlled* claims: continuum extrapolations, finite-volume systematics, renormalization, and reproducible uncertainty quantification.
- **Tractability** corresponds to whether the resulting computation is feasible: sampling/mixing time, variance, and asymptotic cost as one approaches physically relevant regimes.

The point is not that lattice gauge theory is "a programming language," but that it is a structured computational pipeline whose costs and guarantees interact in systematic ways.

**Where the wall appears.** As one pushes toward physically faithful regimes (large volumes, small lattice spacing, light quark masses, and/or finite density), costs can deteriorate sharply. A well-known obstruction is *critical slowing down* toward the continuum limit: autocorrelation times grow rapidly, and the slowest modes are often topological, leading to severe sampling difficulties and "topological freezing" in practice [SSV11]. A second obstruction is the *sign problem* for many fermionic systems (notably at finite chemical potential), for which a generic cure would imply polynomial-time solutions to NP-hard problems; Troyer and Wiese formalize this as an NP-hardness barrier [TW05]. These are not mere implementation bugs: they behave like thermodynamic walls in the computational sense, where increasing model fidelity and maintaining controlled errors drive the required resources out of practical reach.

**EVT-style moral.** The pattern is reminiscent of the earlier "burden-shifting" failure narratives: one can maintain correctness (faithful modeling and controlled systematics) and verifiability (quantified uncertainties) while retaining a rich expressive model class, but the irreducible difficulty reappears as exploding computational cost and method burden. The working physicist pays the price in algorithmic sophistication and computational budgets, even when the underlying formalism remains conceptually clean.

**Relief valves: structure and geometry.** There are, however, important exceptions where new structure changes the computational landscape. The amplituhedron program is a striking example: by exploiting special symmetry and restricting to a highly structured corner (planar $\mathcal{N} = 4$ super Yang–Mills), it recasts certain scattering amplitude computations in terms of positive geometry, dramatically reorganizing and often simplifying what looks combinatorially explosive in standard representations [AHT14]. From the EVT perspective, the lesson is not that "geometry magically defeats complexity," but that *tractability is recovered by moving to a structured subregime*: one trades generality of the theory class for a representation in which verification and computation become more coherent.

This is the sense in which lattice gauge theory supports a broader "computational thermodynamics" viewpoint: when a formalism is simultaneously expressive and verification-driven, some notion of computational feasibility typically becomes the limiting resource, and genuine breakthroughs often come from finding hidden structure that effectively reduces the state space one must traverse.

### 5.2 Numerics and PDEs

Numerical analysis for PDEs offers a near-perfect "computational thermodynamics" analogy: one often trades admissible dynamics, invariants, and guarantees against feasible update rules and stable step sizes.

**A computational reading.** In an EVT-like translation:

- **Expressiveness** corresponds to the class of phenomena one attempts to simulate (shocks, turbulence, stiff source terms, multiscale structure, long-time dynamics).
- **Verifiability** corresponds to enforced properties of the discrete evolution (stability, convergence, conservation laws, monotonicity/entropy conditions, time reversibility, long-time energy behavior).
- **Tractability** corresponds to feasible step sizes and computational cost (time-step restrictions, stiffness barriers, solver complexity).

**Stability walls (CFL-type constraints).** For explicit schemes applied to hyperbolic problems, stability typically demands that the numerical domain of dependence include the physical domain of dependence; this yields Courant–Friedrichs–Lewy (CFL) type constraints that upper-bound the timestep as a function of grid spacing and characteristic wave speed [CFL28]. In practice, increasing resolution or wave speed can force $\Delta t$ to shrink dramatically, so that seeking higher fidelity and stronger stability guarantees drives cost upward.

**Structure-preserving schemes: invariants at a price.** In Hamiltonian and reversible dynamics, *geometric numerical integration* aims to preserve qualitative structure (symplectic form, reversibility, modified energies) over long times [HLW06]. These methods often improve long-time reliability, but they also restrict admissible updates (e.g. to symplectic maps or splitting/composition forms) and may complicate implementation or require implicit solves. Conversely, aggressive damping or stabilization can make computations robust and easy to run, but it typically destroys time-reversal symmetry and fine invariants that one might have wished to verify. The "thermodynamic" moral is the same: enforcing stronger global guarantees generally constrains the update dynamics or shifts cost elsewhere.

### 5.3 Many-body Systems and Field Theory

Many-body physics provides a final instructive analogy. Certain "integrable" corners admit exact charges and factorization, but these corners are structurally special; generic regimes are expressive yet analytically resistant. In the extreme, some semantic properties of dynamics are not decidable uniformly.

**Integrable corners: high verifiability in measure-zero regimes.** Exactly solved models (e.g. those accessible via Bethe ansatz/quantum inverse scattering techniques) exhibit extraordinary structure: large families of conserved quantities, factorized scattering, and exact formulas for spectra and correlation functions [Bax82, KBI93]. In an EVT-like reading, this is a regime where verifiability is unusually high because the dynamics sit inside a rigid algebraic scaffold. But these regimes occupy a thin subset of parameter space: the techniques do not extend uniformly once one perturbs away from integrability.

**Generic regimes: expressiveness without uniform semantic control.** Away from integrable structure, the space of possible many-body behaviors becomes vastly richer (phases, excitations, long-range entanglement structure), but general analytic classification becomes correspondingly difficult. This is not merely a matter of "we have not found the right trick yet": formal barriers are showing that certain global questions about dynamics cannot be decided by a uniform algorithm over all models.

**Undecidability as a semantic barrier.** A sharp illustration is the *spectral gap problem*: given a local, translation-invariant Hamiltonian on an infinite lattice, determine whether it is gapped or gapless in the thermodynamic limit. Cubitt, Pérez-García, and Wolf prove that this problem is undecidable in general [CPGW15]. This is a direct analogue of the "semantic undecidability" pressure that motivates EVT on the PL side: beyond a certain expressive regime, there is no uniform procedure that can always decide nontrivial global properties of the system.

Examples abound. Though a full examination of artificial intelligence systems is beyond the scope of this work, it bears mentioning that recent work by Floridi [Flo25] suggests an analogous tradeoff may even infect the logic of AI systems.

The three case studies above are not meant to "prove" the EVT in physics, but to motivate a broader stance: whenever a formalism is asked to be simultaneously rich (expressive), controlled (verifiable), and usable (feasible in computation and practice), it tends to develop boundary phenomena and recognizable failure modes. In the remainder of this paper, we return to programming languages and make this intuition operational via *resource theory*: a framework for treating expressiveness, verifiability, and their certificates as resources with conversion laws, monotones, and obstructions.

### 5.4 Resource Theory and the Future of the EVT Hypothesis

Resource theories (in the broad sense developed in categorical quantum mechanics and related work) provide a natural formal language for turning "tradeoffs" into mathematically tractable structure: one specifies a class of *free operations*, a notion of *convertibility* between resources, and *monotones* that witness impossibility of certain conversions [CFS16, AC04].[4]

**Future direction.** A resource-theoretic EVT program would proceed in stages.

(a) **Choose the resource objects.** Model a "language design" (or a fragment) as an object carrying: (a) an expressiveness signature (available constructs/abstractions), (b) a verification discipline (types/certificates and their checking relation), and (c) a compilation/evaluation interface. Alternatively, treat *programs and certificates* as the primary resources and languages as resource-generating environments.

(b) **Specify free operations.** Identify transformations that should not count as "consuming" the resources we wish to measure, e.g.:

- conservative syntactic refactorings/definitional expansions,
- macro-eliminations that preserve observational semantics,
- certificate translation that preserves soundness with bounded overhead,
- admissible compiler elaborations and erasures.

The goal is to formalize "same computational content" versus "genuine increase in expressive or verification power."

(c) **Define convertibility preorders.** Expressibility and verifiability subsumption then become instances of resource convertibility:

$$\mathbf{L}_0 \to \mathbf{L}_1 \quad \text{meaning} \quad \mathbf{L}_1 \text{ can simulate/host } \mathbf{L}_0 \text{ under the free operations.}$$

The $E$- and $V$-preorders of this paper should arise as induced orders from this conversion structure.

(d) **Identify monotones ("thermodynamic potentials").** Develop quantities that are monotone under the free operations and therefore certify impossibility results: candidates include bounded derivation/certificate complexity measures, width/structure measures for dependency graphs, and other invariants of the compilation–checking–evaluation pipeline. These monotones would play the role of "resource potentials" that cannot increase for free.

(e) **Recover the frontier and failure modes as no-go theorems.** Recast the EVT boundary as a statement about non-convertibility: certain simultaneous upgrades in expressiveness and verifiability are impossible without consuming additional resources (proof burden, compilation

[4] We treat this as future work; the present paper is intended to stand independently without committing to a specific categorical formalization.

cost, or comprehensibility). The three failure modes (correctness, compilability, comprehensibility collapse) would correspond to distinct ways of relaxing the free-operation constraints or admitting additional "unpaid" resources (oracles, trusted automation, human proof labor).

(f) **Connect to categorical semantics.** Once a stable resource formulation exists, one can integrate it with categorical models of computation and process theories: viewing programs as morphisms, compilation as functors, and certificates as structured morphisms/2-morphisms, with EVT monotones arising from functorial invariants.

**Status.** We view this resource-theoretic program as the most promising route to turning EVT from a collection of boundary phenomena into a compositional calculus of tradeoffs. However, executing it cleanly requires careful choices of free operations and categorical setting, and is best undertaken with specialist input. Accordingly, we leave it as future work and focus here on establishing the order-theoretic and fixed-point/undecidability foundations that any such resource theory must respect.

**Closing Remarks on the EVT Hypothesis.** Proving the expressiveness/verifiability tradeoff for finite state, non-Von Neumann, or geometric models is an exercise suitable for graduate coursework. Extending the result to full Von Neumann architectures and modeling the curve between programming languages is the realm of doctoral dissertations or collaborations between professional research mathematicians and computer scientists with backgrounds in human-computer interaction. But to prove the general EVT Hypothesis at the categorical level, capturing all forms of computation, digital or analog?

That would require mathematical insight worthy of a Fields Medal.

## A Disclosures


### Funding Sources

None.

### Conflicts of Interest

None.


### Data Availability

There is no associated data with this manuscript.


### AI Contributions and Acknowledgments

OpenAI ChatGPT was used as a writing assistant during the development of this work.


## References


[AB11] Samson Abramsky and Adam Brandenburger. The sheaf-theoretic structure of non-locality and contextuality. *New Journal of Physics*, 13:113036, 2011.

[AC04] Samson Abramsky and Bob Coecke. A categorical semantics of quantum protocols. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, pages 415–425. IEEE, 2004.

[AHT14] Nima Arkani-Hamed and Jaroslav Trnka. The amplituhedron. *Journal of High Energy Physics*, 2014(10):030, 2014. arXiv:1312.2007.

[AJM00] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for pcf. *Information and Computation*, 163(2):409–470, 2000.

[Bar84] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[Bar06] Bruno Barras. The guard condition of coq, 2006. Slides/technical note.

[Bax82] Rodney J. Baxter. *Exactly Solved Models in Statistical Mechanics*. Academic Press, 1982.

[Blu67] Manuel Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM*, 14(2):322–336, 1967.

[BMW77] Alan Bawden, David Moon, and Daniel Weinreb. Lisp machine progress report. Technical Report AIM-444, MIT Artificial Intelligence Laboratory, 1977. AI Memo 444.

[Bra13] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[CFL28] Richard Courant, Kurt Friedrichs, and Hans Lewy. über die partiellen differenzengleichungen der mathematischen physik. *Mathematische Annalen*, 100:32–74, 1928.

[CFS16] Bob Coecke, Tobias Fritz, and Robert W. Spekkens. A mathematical theory of resources. *Information and Computation*, 250:59–86, 2016.

[CGL23] Giovanna Kobus Conrado, Amir Kafshdar Goharshady, and Chun Kit Lam. The bounded pathwidth of control-flow graphs. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):292–317, 2023.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, 1988.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[Coq86] Thierry Coquand. An analysis of girard's paradox. In *Proceedings of the Symposium on Logic in Computer Science (LICS)*, pages 227–236. IEEE Computer Society Press, 1986.

[CPGW15] Toby S. Cubitt, David Pérez-García, and Michael M. Wolf. Undecidability of the spectral gap. *Nature*, 528:207–211, 2015.

[Cur42] Haskell B. Curry. The inconsistency of certain formal logics. *The Journal of Symbolic Logic*, 7(3):115–117, 1942.

[CW00] Karl Crary and Stephanie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*, pages 184–198, 2000.

[DL22] Ugo Dal Lago. Implicit computation complexity in higher-order programming languages: A survey in memory of martin hofmann. *Mathematical Structures in Computer Science*, 32(6):760–776, 2022.

[DLR15] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*, pages 140–151, 2015.

[DPJS07] R. Kent Dybvig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, 2007.

[Fel91] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1–3):35–75, 1991.

[Flo25] Luciano Floridi. A conjecture on a fundamental trade-off between certainty and scope in symbolic and generative ai. *Philosophy & Technology*, 38:93, 2025.

[Fut99] Yoshihiko Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[Gim95] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs (TYPES 1994)*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1995.

[Gir72] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. PhD thesis, Université Paris VII, 1972. Foundational work on strong normalization for typed lambda calculi.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[GP96] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[Gre80] Richard D. Greenblatt. A LISP machine. *SIGARCH Computer Architecture News*, 8(3), 1980.

[HAH12] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In *Computer Aided Verification (CAV 2012)*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012.

[HH10] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential: A static inference of polynomial bounds for functional programs. In *Programming Languages and Systems (ESOP 2010)*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.

[HLW06] Ernst Hairer, Christian Lubich, and Gerhard Wanner. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. Springer, 2 edition, 2006.

[How80] William A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980.

[Hue75] Gérard Huet. A unification algorithm for typed $\lambda$-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[Hue76] Gérard Huet. Resolution d'équations dans les langages d'ordre $1, 2, \ldots, \omega$. In *Proceedings of the 2nd International Conference on Automata, Languages and Programming (ICALP)*, 1976. Higher-order unification; semi-decidability and search behavior.

[Hur95] Antonius J. C. Hurkens. A simplification of girard's paradox. In *Typed Lambda Calculi and Applications (TLCA 1995)*, volume 902 of *Lecture Notes in Computer Science*, pages 266–278. Springer, 1995.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[Kam25] M. Benjamin Kaminsky. Geometric computation: Twin helices and the algebra of flow. Sept 2025. Preprint DOI, pending acceptance.

[KBI93] V. E. Korepin, N. M. Bogoliubov, and A. G. Izergin. *Quantum Inverse Scattering Method and Correlation Functions*. Cambridge University Press, 1993.

[Kog91] Peter M. Kogge. *The Architecture of Symbolic Computers*. McGraw-Hill, New York, 1991. Also listed in some catalogs as a 1990 publication/printing.

[KS67] Simon Kochen and Ernst P. Specker. The problem of hidden variables in quantum mechanics. *Journal of Mathematics and Mechanics*, 17(1):59–87, 1967.

[Lan61] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[Lan66] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Moo84] David A. Moon. Garbage collection in a large Lisp system. *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, 1984. Describes GC techniques in the Symbolics 3600 environment.

[NC10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 10th anniversary edition, 2010.

[PD80] David A. Patterson and David R. Ditzel. The case for the reduced instruction set computer. *ACM SIGARCH Computer Architecture News*, 8(6):25–33, 1980.

[Per82] Alan J. Perlis. *Epigrams on Programming*. ACM Press, 1982. Often circulated as "Epigrams in Programming"; includes the "Turing tarpit" epigram.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, Mar 1953.

[Rog67] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw–Hill, 1967.

[SF93] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3–4):289–360, 1993.

[Sha17] Lionel Shapiro. Curry's paradox. Stanford Encyclopedia of Philosophy, 2017. Accessed 2026-03-03.

[Soa87] Robert I. Soare. *Recursively Enumerable Sets and Degrees*. Springer, 1987.

[SSV11] Stefan Schaefer, Rainer Sommer, and Francesco Virotta. Critical slowing down and error analysis in lattice QCD simulations. *Nuclear Physics B*, 845(1):93–119, 2011.

[Ste02] Guy L. Steele. The evolution of Lisp. In Thomas J. Bergin and Richard G. Gibson, editors, *History of Programming Languages, Volume II*. Addison-Wesley, 2002.

[Tai67] William W. Tait. Intensional interpretations of functionals of finite type i. In *The Journal of Symbolic Logic*. 1967. Classic normalization proof technique (Tait's method).

[Tar36] Alfred Tarski. *The Concept of Truth in Formalized Languages*. Oxford University Press, 1936. Translated in *Logic, Semantics, Metamathematics* (1956).

[Tho98] Mikkel Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 1998.

[Tur37] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[TW05] Matthias Troyer and Uwe-Jens Wiese. Computational complexity and fundamental limitations to fermionic quantum monte carlo simulations. *Physical Review Letters*, 94(17):170201, 2005.

[Wel94] J. B. Wells. Typability and type-checking in the second-order lambda-calculus are equivalent and undecidable. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94)*, pages 176–185. IEEE Computer Society, 1994.

[Wel99] J. B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WM81] Daniel Weinreb and David Moon. *The Lisp Machine Manual*. MIT Artificial Intelligence Laboratory, 1981.

[WVAdAE17] Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. A specification for dependent types in haskell. *Proceedings of the ACM on Programming Languages*, 1(ICFP):31:1–31:29, 2017.