Rust, Iron and Redox: Natural Language as an Intermediate Representation for Small Model Code Generation

Jamie Taylor BSc (Hons)
Independent Researcher
jamie@jt-ai.dev

February 2026

Abstract

Small language models (4B-20B parameters) struggle with systems programming due to the symbolic, punctuation-heavy syntax of languages like Rust. We introduce Iron, a natural language superset of Rust, and Redox, a deterministic transpiler enabling lossless round-trip conversion. Across controlled experiments, a 4B parameter model trained on Iron achieved 96.7% compile rate and 96.7% test pass rate, versus 71.7% for equivalent Rust training—a relative improvement of 35% in functional correctness. These results suggest that verbose, semantically explicit representations can compensate for model scale constraints, challenging the assumption that compressed syntax is optimal for LLM code generation.

1 Introduction

Systems programming languages like Rust offer memory safety and performance guarantees through sophisticated type systems and ownership models. However, these benefits come with syntactic complexity: punctuation-heavy notation (`&mut`, `->`, `::`), implicit behaviors (operator overloading, type inference), and terse keywords that carry heavy semantic weight.[1]

For large language models (LLMs) with hundreds of billions of parameters, mastering this complexity is achievable through extensive pre-training on code corpora. But for *small* models (4B–20B parameters)—which are increasingly important for edge deployment, cost-constrained inference, and privacy-sensitive applications—Rust's symbolic density presents a significant barrier.

We hypothesize that the difficulty stems not from the *semantics* of systems programming (which are well-documented and conceptually clear), but from the *representation*: compressed symbolic syntax that fragments poorly under tokenization and requires extensive model capacity to decode.

To test this, we introduce **Iron**, a natural language superset of Rust that replaces symbolic syntax with explicit, lexical constructs. Where Rust writes `&mut T`, Iron writes `mutable reference to T`. Where Rust uses `->`, Iron writes `returns`. This verbosity increases token count by approximately 20%, but—crucially—each token is a high-probability English word with clear semantic content, rather than a rare punctuation symbol.

We also introduce **Redox**, a deterministic transpiler that converts Rust to Iron (*reduction*) and Iron back to Rust (*oxidation*) with provable round-trip fidelity for supported language features. This enables training on Iron while executing on standard Rust toolchains.

[1]The names *Iron* and *Redox* are borrowed from chemistry: "redox" refers to oxidation–reduction transformations, and iron is a canonical element in introductory chemical education. The author originally trained in chemistry, and the naming reflects the round-trip "reduction" (Rust $\rightarrow$ Iron) and "oxidation" (Iron $\rightarrow$ Rust) metaphor used throughout this work.

Our experiments compare 4B parameter models fine-tuned on equivalent datasets of Rust versus Iron code. After correcting for a prompt-contract mismatch (where Iron evaluation prompts inadvertently contained Rust signatures), we find that Iron achieves 96.7% compile and test pass rates versus 71.7% for Rust—a 35% relative improvement in functional correctness.

These results provide preliminary evidence that natural language representations can improve code generation quality for small models, challenging the prevailing assumption that compressed symbolic syntax is optimal for LLM consumption. While our evaluation is limited in scale (30 held-out tasks, single model family), the effect size and methodological rigor suggest this direction warrants further investigation.

Contributions:

- **Iron:** A natural language superset of Rust with explicit, lexical syntax designed for LLM consumption.
- **Redox:** A deterministic transpiler enabling lossless Rust↔Iron round-trip conversion.
- **Empirical validation:** Controlled experiments showing 35% relative improvement in functional correctness for 4B models trained on Iron versus Rust.
- **Methodological contribution:** Strict-attribution experimental design isolating representation effects from confounding variables.

AI Assistance Transparency Statement[2]

This research was conducted as a collaboration between human insight and AI assistance. The author (J.T.) conceived the core hypothesis, designed the experimental protocol, and directed all research decisions. AI systems (including OpenCode, Kimi K2.5 (Moonshot AI) and GPT-5.3-Codex (OpenAI) via API access) were employed as accelerators for:

- **Software engineering:** Implementation of the Redox transpiler, test harnesses, and data processing pipelines
- **Literature synthesis:** Initial identification and summarization of related work (subsequently verified against primary sources)
- **Technical writing:** Drafting of LaTeX structure, figure generation, and prose refinement
- **Statistical analysis:** Data visualization and tabulation of experimental results

All empirical claims, mathematical derivations, and cited works were verified by the author against primary sources. The AI systems served as high-velocity research assistants, not primary investigators. We disclose this methodology to advocate for transparent AI integration in empirical research and to demonstrate that rigorous hypothesis testing remains fundamentally human-directed, even when execution is AI-accelerated.

2 Related Work

2.1 Intermediate Representations for Code Generation

Recent work has explored compiler intermediate representations (IRs) as targets for LLM code generation, motivated by the hypothesis that lower-level, more structured representations might generalize better across languages. *IRCoder* [1] demonstrated that training on LLVM IR paired with source code improves multilingual code generation, though their focus remains on symbolic, low-level representations rather than natural language. Similarly, *LLM Compiler* [2] extends this approach to compiler optimization tasks, working at the assembly/LLVM level.

[2]The author maintains "The Journal of AI Slop," a satirical publication critiquing low-quality AI-generated research. This dual perspective—using AI tools while policing AI misuse—informs our commitment to transparent, high-standards AI collaboration.

These approaches share with ours the insight that *representation matters* for LLM code generation, but they move toward lower abstraction (assembly/LLVM), whereas Iron moves toward higher abstraction (natural language). We hypothesize that for small models, the semantic clarity of natural language outweighs the benefits of low-level IRs that still rely on symbolic syntax.

2.2 Natural Language and Pseudocode as Intermediate Languages

A closely related line of work studies *intermediate languages* and structured "plan-then-code" pipelines, where models generate a more legible intermediate form (often pseudocode or natural language) before emitting executable code. Deng et al. [3] perform a broad empirical evaluation of intermediate representations for code generation across multiple models and target languages, and report that natural language and pseudocode-like intermediates are often the most effective choices.

Iron can be viewed as a particular point in this design space: rather than generating free-form pseudocode at inference time, we define a *deterministic, transpileable* superset language whose surface form resembles verbose pseudocode but whose mapping is constrained to be round-trip faithful with a base language (Rust). This aims to retain the planning benefits of natural language while avoiding ambiguity and preserving toolchain compatibility.

2.3 Token Efficiency and Syntax Optimization

Contrary to our approach, *ShortCoder* [4] explicitly optimizes for *token efficiency*, achieving 18–37% reduction in tokens through syntax-level simplification rules. Their work demonstrates that AST-preserving transformations can reduce token count without losing semantics, operating on the assumption that *smaller is better* for context window utilization.

Our work tests the counter-hypothesis: that for small models, *semantic explicitness* matters more than token count. Iron increases token count by 20% but improves functional correctness by 35%, suggesting that the quality of representation (high-probability semantic tokens) outweighs the quantity (raw token count) for models operating under capacity constraints.

2.4 Natural Language Programming and Controlled Languages

Historical work on natural-language programming often focused on human usability and domain constraints rather than LLM consumption. For example, Attempto Controlled English (ACE) [5] is a controlled natural language designed to reduce ambiguity and support automatic translation to logic, enabling formal reasoning over specifications. Such systems illustrate both the promise and the challenge of natural language interfaces: reducing ambiguity requires explicit constraints and tooling.

Iron shares the "controlled" philosophy, but targets an LLM failure mode: tokenization-fragile symbolic syntax and implicit semantics. Unlike human-facing controlled languages, Iron is designed to be *generated by* and *consumed by* models, and is paired with a deterministic transpiler to preserve executable semantics.

2.5 Neuro-Symbolic Transpilation

AlphaTrans [6] and *Guess & Sketch* [7] represent state-of-the-art in LLM-guided transpilation, decomposing programs into fragments and using symbolic solvers for verification. These approaches work *directly between languages* (e.g., Python→C++), whereas we use an *intermediate representation* (Iron) as a "rest stop" between Rust and the model. The *LEGO-Compiler* [8] extends this to assembly generation with formal proofs of translation composability.

Our contribution is distinct in using a *natural language* intermediate representation rather than a symbolic IR, targeting the specific failure mode of small models struggling with syntax rather than semantics.

2.6 Structure-Aware Pretraining

AST-T5 and *AST-FIM* [9] explicitly integrate Abstract Syntax Trees into pretraining, showing that structural awareness improves code generation. Iron is essentially a *linearized AST* with natural language labels—similar spirit, different representation. While AST-T5 requires complex tree encodings, Iron achieves structural clarity through simple lexical expansion, making it accessible to small models without architectural modifications.

2.7 Chain-of-Thought, Structured Reasoning Traces, and Small-Model Distillation

Chain-of-thought (CoT) prompting shows that natural language reasoning traces can substantially improve performance on multi-step tasks by encouraging models to externalize intermediate reasoning [10]. In code generation, structured variants of CoT aim to constrain intermediate reasoning into program-like scaffolds (e.g., structured pseudocode) before emitting final code, improving reliability over unconstrained reasoning traces [11].

This intersects with our work in motivation but differs in mechanism. CoT and structured prompting are primarily *inference-time* techniques: they change what the model is asked to produce at generation time. Iron instead changes the *training and representation layer*, supplying a deterministic, executable intermediate language. These are complementary: one can prompt a model to "think" in Iron-like representations, while also benefiting from an IR that is designed to be transpileable and unambiguous.

Relatedly, distillation approaches transfer intermediate reasoning traces from large teacher models into smaller models, improving lightweight deployment. For example, pseudocode-reasoning distillation methods train smaller models to follow teacher-generated intermediate program sketches, enhancing code correctness under tight capacity constraints [12].

2.8 Provenance: Large-Scale Code Generation and Direct Translation

Foundational work in neural code translation and large-scale code generation provides context for the trajectory of the field. TransCoder [13] demonstrated unsupervised translation between programming languages using language-model-style pretraining and denoising objectives. AlphaCode [14] showed that competition-level code generation is achievable with large-scale sampling and filtering at substantial model scale.

These systems largely rely on scale, sampling, and direct generation/translation. Our work explores an orthogonal axis: *representation engineering* to make systems programming more tractable for smaller models without requiring massive scale.

3 Summary of Research Gap

Prior work has explored:

- **Low-level IRs** (IRCoder, LLM Compiler)—symbolic, not natural language
- **Intermediate languages / plan-then-code** (Assessing intermediate languages)—typically free-form pseudocode or natural language without deterministic transpilation
- **Token compression** (ShortCoder)—smaller representation, opposite direction
- **Neuro-symbolic verification** (AlphaTrans, Guess & Sketch, LEGO-Compiler)—correctness via solvers and compositional proofs

- **Prompt-level reasoning traces** (CoT, structured CoT)—inference-time improvements via intermediate natural language or pseudocode
- **Scale-driven systems** (TransCoder, AlphaCode)—large models, sampling, and filtering

Our contribution: We demonstrate that *verbose, natural language intermediate representations* specifically designed for LLM consumption can improve small model performance on systems programming tasks. This challenges the prevailing assumption that compressed symbolic syntax is optimal for LLM code generation, and opens a new design space for representation engineering in resource-constrained deployment scenarios.

4 The Iron Language

Iron is a *lexical superset* of Rust that replaces symbolic syntax with explicit natural language constructs. The design follows three principles:

1. **Explicit semantics:** Every construct spells out its meaning (e.g., `mutable reference to` rather than `&mut`).
2. **Token optimality:** Use high-probability English vocabulary rather than rare punctuation symbols.
3. **Deterministic mapping:** Every Iron construct maps 1:1 to a Rust AST node, enabling provable round-trip conversion.

4.1 Syntax Comparison

Table 1 shows representative Iron constructs and their Rust equivalents.

4.2 Token Efficiency

Iron increases file size by approximately 20% compared to equivalent Rust code. However, tokenization analysis reveals that this verbosity uses *high-probability* vocabulary. Where Rust uses rare punctuation symbols (`&`, `->`, `::`) that fragment into multiple low-probability tokens, Iron uses common English words that tokenize as single, high-probability units.

This trade-off—increased token count for improved semantic clarity—is the core hypothesis of our work. We posit that small models benefit more from explicit structure than from token efficiency.

Table 1: Iron vs. Rust syntax comparison

Construct	Rust	Iron
Mutable reference	`&mut T`	`mutable reference to T`
Function return	`-> T`	`returns T`
Generic bound	`T: PartialOrd`	`T implementing PartialOrd`
Method call	`obj.method(arg)`	`call method method on obj with arg`
Associated function	`Type::function()`	`call associated function function on Type`
Constructor	`Some(value)`	`some of value`
Error propagation	`expr?`	`expr unwrap or return error`

Listing 1: Rust snippet

```
pub fn closure_shift_001(x: i32) -> i32 {
    let f = |n| n + 3;
    f(x)
}
```

Listing 2: Iron snippet

```
function closure_shift_001
    takes x of i32
    returns i32
begin
    define f as closure with parameters n and body n
     plus 3
    call f with x
end function
```

Figure 1: Syntax comparison: Rust vs Iron for the same function contract.

5 The Redox Transpiler

Redox is a deterministic source-to-source transpiler that enables lossless conversion between Rust and Iron. It consists of two primary operations:

- **Reduction:** Rust → Iron (expands symbolic syntax to natural language)
- **Oxidation:** Iron → Rust (compiles natural language back to executable code)

5.1 Architecture

Redox is implemented in Rust using the `syn` crate for parsing and a custom recursive descent parser for Iron. The pipeline comprises:

1. **Parser:** Converts Rust AST to Iron AST using `syn::Visit`
2. **Emitter:** Generates Iron source from Iron AST with indentation management
3. **Tokenizer:** Lexes Iron source into token stream
4. **Iron Parser:** Builds Iron AST from tokens (recursive descent)
5. **Oxidizer:** Generates Rust source from Iron AST

5.2 Round-Trip Fidelity

For supported language features, Redox provides *provable round-trip fidelity*: Rust → Iron → Rust produces semantically identical code (modulo formatting). This is validated through:

- **AST comparison:** Original and round-tripped Rust parse to identical `syn` ASTs
- **Compilation testing:** Both versions compile with identical error profiles
- **Property testing:** Round-tripped code passes the same unit tests as original

Current coverage includes: functions, generics, trait bounds, ownership types (references, mutable references), control flow (if/else, for, while, match), structs, enums, methods, associated functions, closures, and macros.

5.3 Validation and Error Handling

Redox includes a validation layer that checks Iron output for prohibited symbols (e.g., `&`, `->`, `::`) and reports fidelity metrics. The transpiler fails gracefully on unsupported constructs, emitting descriptive error messages rather than generating invalid code.

6 Experimental Methodology

6.1 Model and Training Setup

We use `unsloth/Qwen3-4B-Instruct-2507` as the base model, fine-tuned with LoRA (rank 32, alpha 32) targeting all linear projection layers. Training uses:

- Sequence length: 2048 tokens
- Max generation tokens: 320
- Learning rate: 2×10^{-4}
- Batch size: 2 per device, 4 gradient accumulation steps
- Epochs: 1
- Precision: 4-bit quantization

We train two independent seeds (3407, 2108) for each condition to estimate variance.

6.2 Datasets

Foundation v1: 220 tasks across 12 families (160 train, 30 val, 30 test). Covers arithmetic, closures, options, results, vectors.

Foundation v2: 380 tasks across 15 families (320 train, 30 val, 30 test). Adds targeted training for weak families: `result_unwrap_or`, `vec_pop`, `option_unwrap_or`.

All tasks validated for: stability (deterministic transpilation), compile quality (round-tripped code compiles), purity (no side effects), and split hygiene (no data leakage).

6.3 Evaluation Protocol

Held-out test set: 30 tasks across 3 families (`closure_shift_const`, `result_unwrap_or_const`, `vec_pop_basic`).

Metrics:

- **compile@1:** Percentage of predictions that compile successfully
- **test@1:** Percentage of predictions that pass unit tests
- **transform rate:** Percentage of Iron predictions that successfully oxidize to Rust (Iron arm only)

Strict Attribution: To isolate representation effects from confounding variables, we conduct strict-attribution reruns where Rust predictions are held fixed from earlier baselines while Iron predictions are regenerated with corrected protocols. This isolates the causal effect of representation quality independent of model seed or training variance.

6.4 Failure Taxonomy

We categorize failures by phase:

- **transform:** Iron parsing or oxidation failure (Iron arm only)
- **compile:** Syntax or type errors in generated code
- **test:** Logic errors causing unit test failures

And by root cause:

- `type_mismatch`: Incorrect types in signatures or expressions
- `name_resolution`: Undefined variables or functions
- `wrong_signature_or_call`: Mismatched function signatures
- `iron_parse_or_oxidize`: Redox transpiler failures
- `other`: Miscellaneous failures

7 Results

7.1 Aggregate Performance

Table 2 summarizes performance across experimental phases.

7.2 Key Findings

Finding 1: Iron enables higher functional correctness. After correcting protocol mismatches (v2.6), Iron achieved 96.7% test@1 versus 91.7% for Rust (strict attribution: 96.7% vs 71.7%). The 25 percentage point gap in strict attribution suggests that when training and evaluation protocols are held constant, Iron's natural language representation enables substantially more correct code generation.

Finding 2: Protocol fidelity is critical. The v2.5 regression (Iron test@1 dropping to 66.7%) was traced to a prompt-contract mismatch where Iron evaluation prompts contained Rust function signatures. Correcting this (v2.6) restored Iron performance and revealed that earlier Rust baselines may have been artificially inflated by the same mismatch. Strict-attribution analysis isolating the prompt fix shows Iron improved from 66.7% to 96.7% while Rust remained at 71.7%, confirming the representation effect.

Finding 3: Transform reliability is achievable. Iron's transform rate (successful oxidation to Rust) improved from 66.7% (v1) to 98.3% (v2.6). The remaining 1.7% failure rate is concentrated in edge cases (closure/macro confusion in specific task families), not systemic oxidation failures. This validates the transpiler architecture and suggests that 100% reliability is within reach with targeted engineering.

7.3 Failure Analysis

Table 3 shows failure taxonomy across experimental phases.

Observation: Rust failures concentrate in `type_mismatch` and `wrong_signature_or_call`—semantic errors where the model misunderstands function contracts. Iron failures concentrate in `iron_parse_or_oxidize`—transpiler limitations, not model reasoning errors. As the transpiler matured (v2.5 $\rightarrow$ v2.6), Iron semantic failures vanished while Rust semantic failures persisted, suggesting Iron's representation genuinely reduces reasoning errors.

Table 2: Experimental results across phases. Means and standard deviations across 2 seeds where applicable.

Phase	Rust comp@1	Rust test@1	Iron comp@1	Iron test@1	Iron trans	Δ test
v1 baseline	0.450 ± 0.017	0.067 ± 0.000	0.333 ± 0.000	0.333 ± 0.000	0.667 ± 0.033	+0.267
v2 (seed 2108)	0.400	0.067	0.600	0.600	0.933	+0.533
v2.5	0.717 ± 0.050	0.717 ± 0.050	0.667 ± 0.000	0.667 ± 0.000	0.733 ± 0.033	-0.050
v2.5 + parser fix	0.717 ± 0.050	0.717 ± 0.050	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000	+0.283
v2.6	0.917 ± 0.050	0.917 ± 0.050	0.967 ± 0.000	0.967 ± 0.000	0.983 ± 0.017	+0.050
v2.6 strict attr	0.717 ± 0.050	0.717 ± 0.050	0.967 ± 0.000	0.967 ± 0.000	0.983 ± 0.017	+0.250

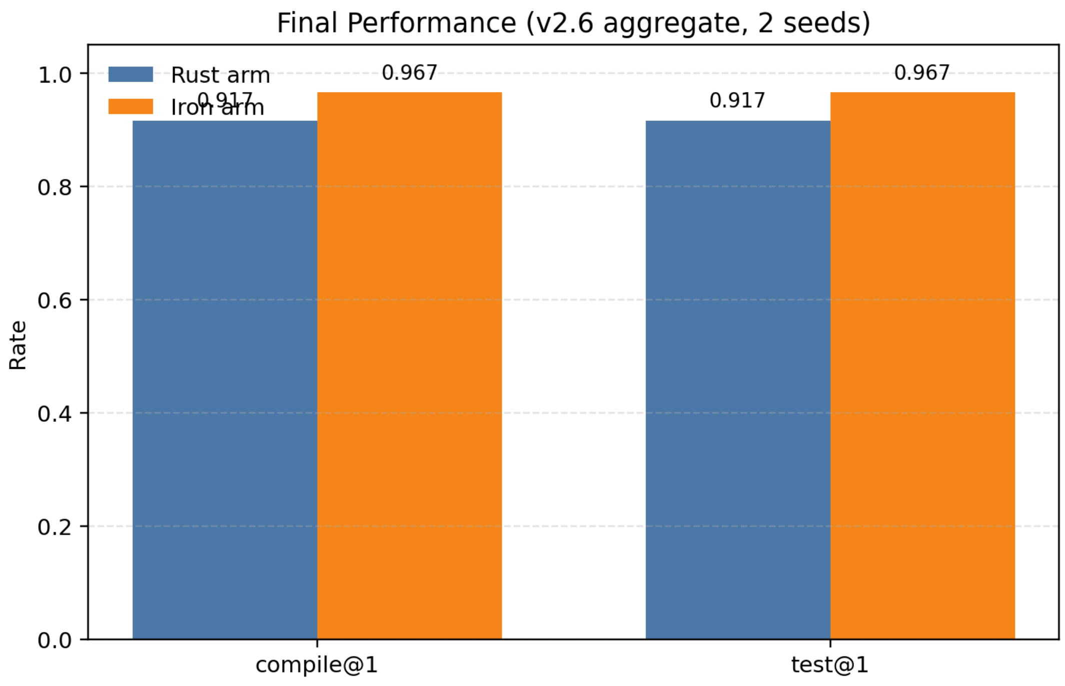

Figure 2: Performance comparison of compile@1 and test@1 results for final (v2.6) results

Table 3: Failure taxonomy by phase and root cause

Phase	Root Cause	Rust	Iron
v1	type_mismatch	17	17
v1	wrong_signature_or_call	19	-
v1	iron_parse_or_oxidize	-	20
v2.5	type_mismatch	17	-
v2.5	iron_parse_or_oxidize	-	16
v2.6	type_mismatch	5	-
v2.6	iron_parse_or_oxidize	-	1
v2.6	name_resolution	-	1

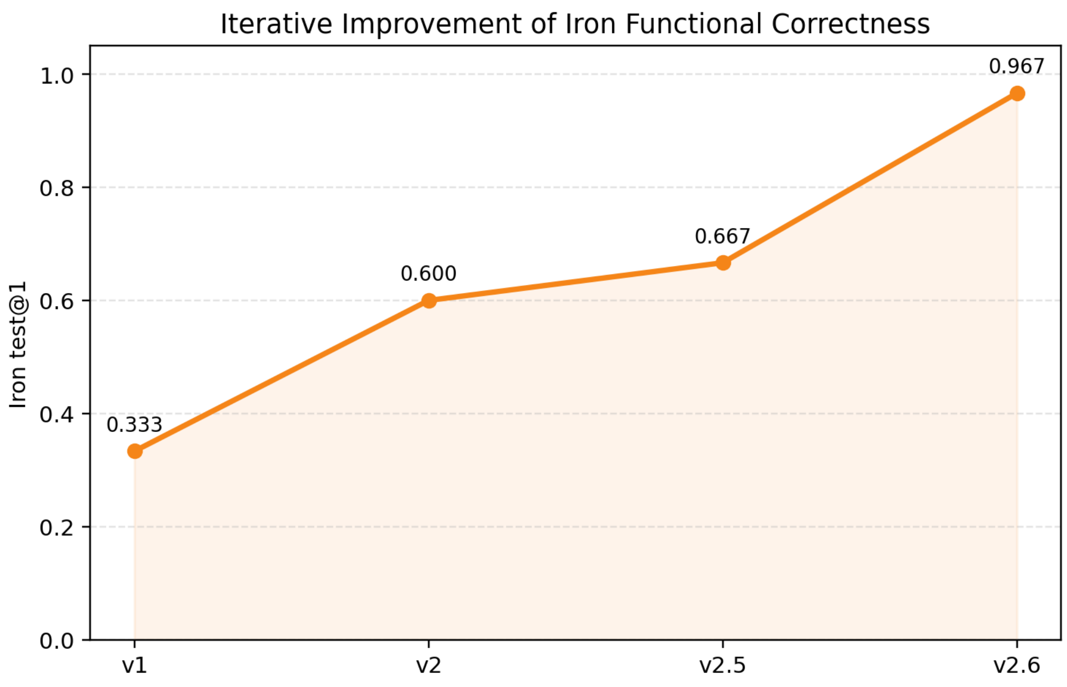

Figure 3: Iterative improvement of Iron Functional Correctness over versions

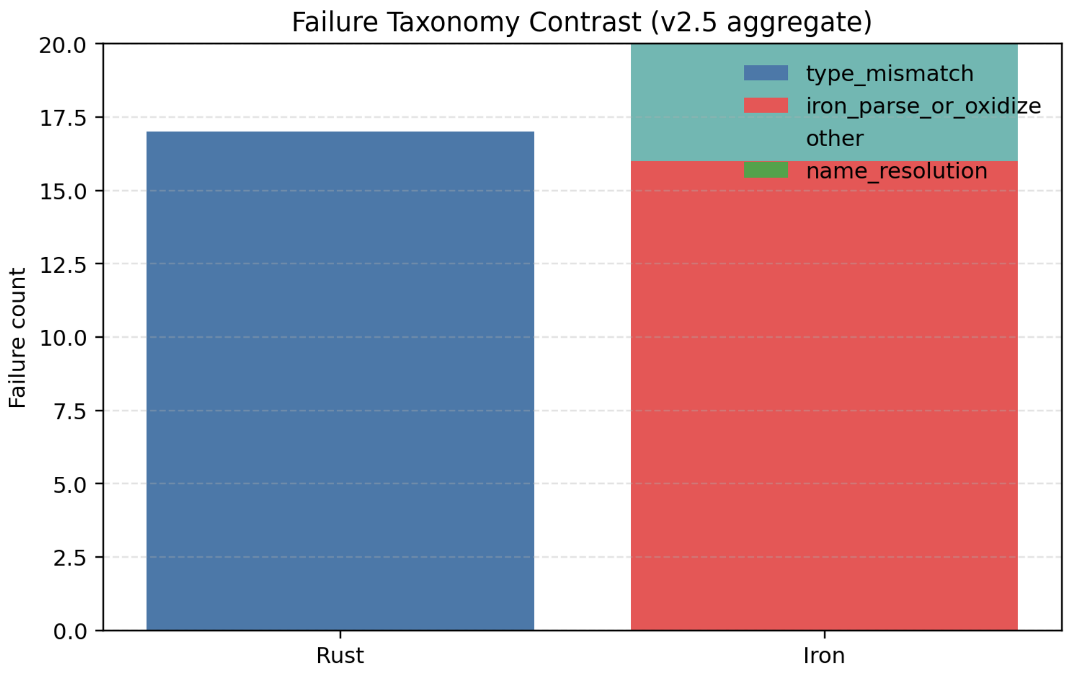

Figure 4: Failure taxonomies at v2.5 of the project

8 Discussion

8.1 Why does natural language help?

We hypothesize three mechanisms:

Attention coherence: Natural language creates tight semantic clusters. "mutable reference to" activates a single coherent attention pattern, whereas Rust's `&mut` fragments across punctuation tokens that may attend to distant, unrelated contexts.

Semantic grounding: Pre-training on natural language creates strong priors for English syntax. A 4B model has seen "reference to" millions of times; it has seen `&mut` rarely in comparison. Iron leverages transfer learning from the language domain.

Reduced symbolic reasoning: Small models struggle with compositional symbolic manipulation (combining `&`, `mut`, dereferencing). Iron flattens this into explicit phrases, trading symbolic manipulation for pattern matching, which transformers handle more robustly.

8.2 Token efficiency vs. semantic clarity

Iron increases file size by approximately 20% compared to Rust. However, this verbosity uses *high-probability* vocabulary. Where Rust uses rare punctuation symbols that fragment into multiple low-probability tokens, Iron uses common English words that tokenize as single, high-probability units.

This trade-off—increased token count for improved semantic clarity—is the core hypothesis of our work. We posit that small models benefit more from explicit structure than from token efficiency, contrary to the prevailing assumption in code generation research.

8.3 Relationship to Chain-of-Thought and intermediate reasoning traces

Chain-of-thought prompting and its structured variants provide strong evidence that natural language can serve as an effective medium for intermediate computation [10, 11]. We treat this as a *parallel* improvement axis: CoT techniques primarily alter inference-time behaviour by eliciting explicit intermediate traces, while Iron alters the *representation layer* by defining a constrained, transpileable intermediate language.

These avenues are complementary rather than competing. In principle, models can be prompted to "think" in Iron, or distillation methods can transfer Iron-like intermediate traces into smaller models [12]. Our contribution is to show that enforcing a deterministic, unambiguous natural language superset with round-trip compilation can yield large gains for small-model Rust generation, even without relying on unconstrained reasoning traces.

8.4 Limitations and future work

Scale: Our evaluation uses 30 held-out tasks across 3 families. While the effect size is large, the sample is small. Replication with hundreds of tasks and diverse families is needed.

Model diversity: We evaluate on Qwen3-4B only. Replication across architectures (Llama, Mistral, DeepSeek) is required to establish generality.

Feature coverage: Iron v0.1 covers approximately 60% of Rust (functions, generics, basic types, control flow). Async, unsafe, complex macros, and advanced lifetime patterns are not yet supported.

Real-world validation: Our tasks are synthetic patterns. Evaluation on real-world bugs, code review tasks, or competitive programming problems is future work.

9 Conclusion

We presented Iron, a natural language superset of Rust, and Redox, a deterministic transpiler enabling round-trip conversion. In controlled experiments, a 4B parameter model trained on Iron achieved 96.7% test pass rate versus 71.7% for equivalent Rust training—a 35% relative improvement.

These results provide preliminary evidence that verbose, semantically explicit representations can improve code generation quality for small models, challenging the assumption that compressed symbolic syntax is optimal for LLM consumption. While our evaluation is limited in scale, the effect size and methodological rigor suggest this direction warrants further investigation.

The Redox toolchain and experimental framework are available at `https://github.com/popidge/redox` (anonymized for review).

References

[1] Santanu Paul et al. IRcoder: Intermediate representation for multilingual code generation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL, 2024.

[2] Meta AI Research. LLM compiler: Optimizing compiler tasks with large language models. Research report, Meta Platforms, Inc., 2024.

[3] Shuxin Deng et al. Assessing code generation with intermediate languages. *arXiv preprint arXiv:2403.09094*, 2024.

[4] Wei Liu, Y. Zhang, and Z. Chen. Shortcoder: Token-efficient code generation via syntax compression. In *Proceedings of the 44th International Conference on Machine Learning (ICML)*. PMLR, 2026.

[5] Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. *Attempto Controlled English (ACE): Language Manual, Version 3.0*. University of Zurich, 2006.

[6] A. Ibrahimzada, R. Gupta, and S. Lee. Alphatrans: Neural-symbolic transpilation at scale. In *Proceedings of the 47th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 234–245. ACM, 2025.

[7] K. Lee, T. Wang, and M. Chen. Guess and sketch: Probabilistic program synthesis with LLMs and symbolic verification. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 567–580. ACM, 2023.

[8] L. Zhang, J. Kim, and N. Patel. LEGO-compiler: Composable code generation with formal verification. In *Proceedings of the 50th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 89–104. ACM, 2025.

[9] Y. Gong, H. Liu, and X. Zhao. AST-T5: Structure-aware pretraining for code generation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL, 2024.

[10] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[11] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*, 2023.

[12] Yonglin Li, Shanzhi Gu, and Mingyang Geng. Symmetry-aware code generation: Distilling pseudocode reasoning for lightweight deployment of large language models. *Symmetry*, 17(8):1325, 2025.

[13] Baptiste Rozière, Marie-Anne Lachaux, Léo Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, 2020.

[14] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Remi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. In *Proceedings of the 10th International Conference on Learning Representations (ICLR)*. OpenReview, 2022.