gp-λ: A Model-Centric Programming Language for Generative AI Systems

Reinout Wijnholds

Abstract

Large Language Models (LLMs) are increasingly writing software, yet current programming languages are designed for human readability rather than machine-generation. This gap motivates **gp-λ**, a new programming language centered on *model executability* rather than human comprehension. Gp-λ provides a compact, formal syntax and semantics tailored for LLM code generation pipelines. It abandons extraneous human-oriented syntax in favor of structures that are easily parsed, checked, and verified by machines. Key features include a strict type-and-effect system (distinguishing pure computations from those with I/O or filesystem/network effects), resource-aware constructs like `cost{}` blocks to enforce time/space bounds, and inline `proof{}` blocks where formal proofs of correctness obligations can be discharged. We present a complete formal definition of gp-λ's syntax, operational semantics, and type system, including effect tracking and cost semantics. We mechanize gp-λ's metatheory in Coq, proving standard safety properties (progress and preservation) as well as a novel *cost soundness* theorem guaranteeing that programs respect statically-declared resource budgets. Example gp-λ programs illustrate its expressiveness: from pure recursive functions to effectful I/O tasks with statically tracked side-effects, and loops with proven runtime bounds. We compare gp-λ to state-of-the-art safe and verified languages (Liquid Haskell, Rust (RustBelt core), and F*), finding that gp-λ achieves competitive verification power with significantly more concise, LLM-friendly code. Finally, we discuss how gp-λ aligns with LLM-based development workflows—enabling on-the-fly verification and cost checking during code generation—and the implications for future AI-centric software engineering. Gp-λ represents a step towards *model-generated, verifiably correct, and resource-efficient* software.

1 Introduction

Recent advances in generative AI have led to LLMs capable of producing substantial amounts of source code. Studies indicate that as of 2025, up to 97% of software developers incorporate generative AI into their workflow, and roughly 30% of code being written is AI-generated:contentReference[oaicite:0]index=0. While this trend promises productivity gains, it also exposes critical issues: the code generated by LLMs often contains vulnerabilities:contentReference[oaicite:1]index=1 or other defects that a human might catch only after deployment. For example, an empirical security analysis found that LLMs frequently generate insecure code, highlighting a need for stronger safety guarantees in AI-generated software:contentReference[oaicite:2]index=2. These challenges call for rethinking programming language design in the context of AI-as-coder.

Traditional programming languages were created with human developers in mind: they prioritize readability, expressiveness, and manual maintainability. However, as we enter an era where the "programmer" is increasingly an AI model, the criteria for an effective language shift significantly. As Jones argues, the next generation of programming languages will not evolve towards natural human language, but rather toward more formal, machine-centric paradigms driven by (1) how accurately an LLM can generate code, (2) how efficiently that code can be verified for its intended purpose, (3) how easily humans can interpret and integrate it, and (4) how efficiently it executes:contentReference[oaicite:3]index=3:contentReference[oaicite:4]index=4. In particular, dynamically-typed, informally-specified languages like Python—designed for rapid human scripting—may be ill-suited when AI is producing the code, since they lack the static verification features needed to catch LLM mistakes:contentReference[oaicite:5]index=5. As an example, Python's dynamic semantics impose overhead and allow subtle runtime errors, which an LLM may not properly account for; thus experts have speculated whether a more strongly-typed or mathematically structured language should replace Python in the LLM era:contentReference[oaicite:6]index=6. More broadly, conventional languages "are mostly designed to interact better with humans, whereas in the future, programming languages need to address how to communicate better with large models":contentReference[oaicite:7]index=7.

Early vision papers have posited this paradigm shift from human-readability to *LLM-executability* in programming languages. For instance, John et al. outline the challenges large models pose to current languages—security vulnerabilities in model-generated code, and hallucinated or contextually inconsistent completions—and urge that future languages incorporate features to mitigate these issues:contentReference[oaicite:8]index=8:contentReference[oaicite:9]index=9.

They argue an AI-centric language should emphasize three characteristics: ease of automated testing, rapid static analysis, and intrinsic security safeguards:contentReference[oaicite:10]index=10:contentReference[oaicite:11]index=11. In essence, the language itself must assist the model in avoiding mistakes, by making program properties (like types, resource usage, and invariants) explicit and checkable. These recommendations align with independent studies on LLM coding errors: for example, package hallucination vulnerabilities (where an LLM suggests a nonexistent library that attackers can hijack) can be curbed by enforcing that all dependencies are declared and verified:contentReference[oaicite:12]index=12:contentReference[oaicite:13]index=13. Taken together, these insights motivate a new programming language paradigm where formal rigor and machine verifiability are foremost, even at the expense of human ergonomics.

In this paper, we introduce **gp-λ** (pronounced "GP-lambda"), a programming language designed from the ground up for generative AI systems rather than human programmers. Gp-λ draws inspiration from classical λ-calculus and modern type theory, but repurposes them in a model-centric fashion. The guiding principle is to make the language as *unambiguous, structured, and amenable to automation* as possible. Concretely, gp-λ has the following distinguishing features:

- A minimalistic, uniformly structured syntax that reduces noise and redundancy, aiding both parsing and generation by LLMs. The grammar favors concise keywords and symbolic notations over natural-language-like constructs, resulting in code that is token-dense and focused on semantics (Section 2). This token compactness aligns with the need for LLMs to stay within context window limits while generating code.

- A **type-and-effect system** that classifies computations as Pure (mathematical functions with no side effects), IO (possibly performing console or user-interaction effects), or more finely, FS/Net (accessing filesystem or network). This effect tracking is baked into function types and enforced by the type checker (Section 2.2). By making side effects explicit, gp-λ allows an LLM (and a downstream static analyzer) to understand the scope of a code fragment's interactions with the external world, addressing the "illusion" problem of missing context in completions:contentReference[oaicite:14]index=14. The effect system draws on ideas from languages like Haskell and F*'s effect polymorphism:contentReference[oaicite:15]index=15 but is tailored to be simple enough for automated reasoning.

- **Resource-aware constructs** that enable reasoning about time and space complexity at the language level. In particular, gp-λ provides a special `cost{}` block, in which the programmer (or the code-generating model) annotates a block of code with an expected resource cost (e.g., an upper bound on steps or memory usage). The type system, via a form of static cost analysis, checks that the code within the block does not exceed the claimed cost (Section 2.3). This feature addresses the call for security and efficiency: it prevents an LLM from unknowingly producing extremely slow or memory-intensive code and enables the automatic certification of complexity bounds. Our approach is informed by Automatic Amortized Resource Analysis (AARA):contentReference[oaicite:16]index=1 a type-based technique for inferring symbolic resource bounds, which we integrate into gp-λ's analysis. To our knowledge, gp-λ is the first language to expose a cost verification construct explicitly intended to guide and check LLM-generated code.

- **Inline proof obligations** through a `proof{}` block syntax. Gp-λ treats certain correctness properties (e.g., functional correctness conditions, loop invariants, or user-specified assertions) as logical propositions that must be proven by providing a proof term or script inside a `proof{}` block. These proof blocks are checked by a proof assistant (or possibly by an LLM specialized in proof) and are erased from the executable code. This design is akin to the approach of Coq or F* where one can embed proofs within programs:contentReference[oaicite:18]index=18, but gp-λ makes the mechanism extremely direct: if a proof is required at a program point, the code literally contains a `proof{}` placeholder that the model must fill with a derivation. By forcing the LLM to confront proof obligations, we aim to eliminate guesswork in critical algorithmic reasoning; the model either produces a verifiable proof or the code will not type-check. This helps ensure reliability of AI-generated solutions, as those proofs can guarantee properties that testing might miss.

We have formalized the core of gp-λ to ensure these features are not only intuitively useful but also rigorously sound. In Section 3, we describe the formal semantics of gp-λ, including a small-step operational semantics extended with a cost metric, and a corresponding Coq mechanization. We prove key metatheoretical results: *type soundness* (well-typed programs cannot go wrong at runtime, i.e. no type errors or effect violations will occur, as per preservation and progress theorems) and *cost soundness* (the evaluation cost of any `cost{}` block respect its annotated

bound). The cost soundness property is inspired by the soundness proofs in AARA literature which show that typable programs respect inferred resource bounds:contentReference[oaicite:19]index=19; here, we adapt those techniques to an explicitly annotated setting and mechanize the proof. We also ensure that proof blocks do not affect execution (they can be erased) yet any proved properties hold in the operational semantics (consistency of the proof system).

In Section 4, we illustrate gp-λ with example programs. Despite its machine-oriented nature, gp-λ is expressive enough to write typical algorithms. We show: (a) a pure recursive function on numbers (classic factorial), (b) an I/O routine with effect tracking, (c) a loop that computes a sum with an accompanying cost verification that it runs in linear time, and (d) a scenario where a resource-bound violation is caught at compile-time. These examples demonstrate how a generative model might use gp-λ to produce code that comes with built-in guarantees (e.g., total absence of buffer overflows or excessive runtime).

In Section 5, we compare gp-λ to several existing languages and verification systems. **Liquid Haskell** augments Haskell with refinement types to prove correctness properties in a lightweight way:contentReference[oaicite:20]index=20. We discuss how gp-λ differs by targeting a different audience (LLMs vs. human programmers) and by including cost analysis and effects natively. **Rust** (and its formal core calculus used in RustBelt) provides strong safety guarantees through ownership; we contrast this with gp-λ's approach of requiring proofs for safety-critical sections instead of zero-cost abstractions, and note that Rust's assurances (formally proven by RustBelt:contentReference[oaicite:21]index=21) are complementary to gp-λ's goals. **F*** is perhaps the closest in spirit: a dependently-typed, verification-oriented language with an SMT-backed proof sidecar:contentReference[oaicite:22]index=22. We compare gp-λ's design trade-offs (simplicity and token-economy vs. expressive power) with those of F*, and highlight that gp-λ aims to be significantly more compact in syntax, which is advantageous for large models that must predict code one token at a time.

Section 6 provides a broader discussion on how gp-λ fits into the emerging AI code generation pipeline. By making program structure and correctness explicit, gp-λ can synergize with LLM-based tooling in ways traditional languages cannot. For example, an LLM-empowered IDE could use gp-λ's effect annotations and cost blocks to give immediate feedback or corrective hints to the model as it writes code (similar to how MoonBit's IDE performs real-time static analysis during code completion:contentReference[oaicite:23]index=23). We also consider the implications for fine-tuning LLMs: training a model on

formally verified gp-λ code could encourage the model to internalize logical reasoning steps, potentially reducing hallucinations and errors. In terms of compiler design, gp-λ acts as a high-level IR where many correctness guarantees are already certified at the front-end, simplifying the compiler's job to preserve those guarantees and perhaps enabling new optimizations (for instance, using cost annotations to inform scheduling or parallelization decisions). Finally, we outline future directions, including integrating automatic proof search (so that the LLM doesn't have to manually prove trivial obligations) and exploring subset variations of gp-λ as target languages for neural program synthesizers.

We conclude in Section 7 that gp-λ provides a promising foundation for model-generated software. By reimagining language design for an AI agent instead of a human, we can achieve verifiability and efficiency at generation time, addressing both the security and "illusion" issues identified in current AI coding practices:contentReference[oaicite:24]index=24:contentReference[oaicite:25]index=25. Gp-λ demonstrates that it is feasible to have a programming language that is *by the model, for the model*: it meets the model halfway by being formal and structured, and in return yields software that is safer and more predictable. We hope this work spurs further research at the intersection of programming languages, formal methods, and AI, moving us closer to a future of reliable AI-generated software.

2 The gp-λ Language

In this section, we formally define gp-λ's core language. We present its syntax (with a BNF grammar), typing rules (including the effect and cost systems), and a sketch of the operational semantics. Our design goal is a language that is minimal yet sufficiently expressive, with all critical behaviors made explicit in the syntax or types.

2.1 Syntax

The syntax of gp-λ is summarized in Figure 1. The language is expression-oriented and functional, with variables, lambda abstractions, and application forming the basis (a λ-calculus core). In addition, it provides conditional expressions, let-binding, sequence, and constants (like integers, booleans, etc.). Distinctive to gp-λ, we include two special block constructs: a **cost block cost**$\{e\}$ and a **proof block proof**$\{e_{prf}\}$. We also incorporate a simple imperative flavor (mutable variables and loops) in the surface syntax for convenience, but these can be encoded into the pure core if needed.

$$
\begin{aligned}
\textit{Base types } \iota &::= \mathbf{Int} \mid \mathbf{Bool} \mid \mathbf{String} \mid \cdots \\
\textit{Effects } \epsilon &::= \mathbf{Pure} \mid \mathbf{IO} \mid \mathbf{FS} \mid \mathbf{Net} \\
\textit{Types } \tau &::= \iota \mid \mathbf{unit} \mid \tau_1 \xrightarrow{\epsilon} \tau_2 \mid \mathbf{Proof} \\
\textit{Expr. } e &::= x \mid c \mid \lambda(x{:}\tau).\, e \mid e_1\, e_2 \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \\
&\quad \mid \ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{cost}\{e\} \mid \mathbf{proof}\{e_{prf}\} \mid () \\
&\quad \mid \ \mathbf{var}\ x = e_1;\ e_2 \mid x := e_1 \mid \mathbf{while}\ e_1\ \{e_2\} \mid \mathbf{print}(e)
\end{aligned}
$$

Figure 1: Gp-λ syntax. We write c for constants and x for identifiers. Effects ϵ annotate function arrow types and classify effectful operations. **unit** is the singleton type (with value ()). A few imperative constructs (mutable variable and loop) are included for illustration, although not strictly part of the core calculus.

Every function type in gp-λ is annotated with an *effect* ϵ indicating the maximal effect of that function. For example, a function of type $\mathbf{Int} \xrightarrow{\mathbf{Pure}} \mathbf{Int}$ is a pure function from Int to Int (it cannot perform I/O or other side effects), whereas $\mathbf{String} \xrightarrow{\mathbf{IO}} \mathbf{unit}$ would be a function that takes a string and returns unit, potentially performing I/O (e.g., printing to console). Effects form a lattice of sorts, with **Pure** as the most restrictive (no side effects) and **IO**, **FS**, **Net** as increasingly powerful effects. We consider **FS** and **Net** as subcategories of I/O that specifically involve filesystem or network operations respectively; for simplicity in this core calculus, we sometimes treat **IO** as encompassing both. The effect system is inspired by effect polymorphism in F*:contentReference[oaicite:26]index=26 and traditional monadic I/O in Haskell, but here it is explicit in the type to simplify reasoning by an LLM or checker.

The $\mathbf{cost}\{e\}$ construct is an expression that carries an implicit resource bound annotation. In practice, one would write $\mathbf{cost}^C\{e\}$ (specifying a cost metric C such as a number of steps or an asymptotic form); we treat C as meta-data and often omit it in the grammar for brevity. Intuitively, $\mathbf{cost}\{e\}$ means that the enclosed expression e is claimed to execute within a certain cost budget. The type system will enforce this claim by static analysis (Section 2.3). If the claim cannot be verified, the program is rejected as ill-typed (the AI model would then need to adjust the code or cost annotation).

The $\mathbf{proof}\{e_{prf}\}$ construct is a block that contains a proof term or proof script, e_{prf}, intended to establish some logical property required at that

point. For example, if the program needs to prove that a loop invariant holds or that a function meets its specification, a **proof**{} block would contain the derivation. The proof language e_{prf} can be thought of as a sequence of tactics or a proof term in a logical calculus; for our purposes, we won't fix a specific proof language (it could be a sequence of Coq tactics or a lambda-term in a proof system). What matters is that **proof**$\{e_{prf}\}$ has type **Proof** if e_{prf} is a valid proof of the obligation in context. Proof blocks do not produce computational effects or values (aside from a trivial **unit** or **Proof** token); they are purely for verification. At runtime, proof blocks are erased. We include **Proof** as a special type of proof objects (similar to how Coq treats propositions in Prop, which do not exist at runtime).

Finally, we include a basic mutable variable and loop in the extended syntax (as sugar on top of the core). This is to demonstrate that gp-λ can handle typical imperative patterns (useful since many AI-generated code pieces involve loops or mutable state). Under the hood, one could desugar these to pure recursive functions or state monads, but here we treat them informally to keep examples straightforward.

2.2 Type System and Effects

The typing judgment in gp-λ is of the form $\Gamma \vdash e : \tau \ [\epsilon]$, which reads: "under context Γ, expression e has type τ and has an effect (at most) ϵ." The context Γ assigns types to free variables. We use a notation where the effect of an expression is tracked alongside the type. For value expressions (which produce no side effects), the effect will be **Pure**. For expressions that perform I/O or other effects, the judgment will carry a correspondingly higher effect.

Some representative typing rules are shown in Figure 2. We highlight the handling of effects and the special constructs:

Functions and application. In rule T-Lam, a lambda abstraction $\lambda(x : \tau_1).e$ is given the function type $\tau_1 \xrightarrow{\epsilon} \tau_2$, where ϵ is the effect of the function body e. The function itself is a value, hence typing it is a pure act (no side effects just by creating a closure). Rule T-PureApp handles the application of a pure function: if e_1 is a function with effect **Pure** from τ_1 to τ_2, and e_2 is a pure argument of type τ_1, then $e_1\ e_2$ is pure and has type τ_2. Rule T-IOApp handles the case where the function expects or produces an effect. If e_1 has type $\tau_1 \xrightarrow{\mathbf{IO}} \tau_2$ (meaning calling it may do I/O) and e_2 is its argument, then the result is an expression of type τ_2 that is considered to have effect **IO**. In general, any function with effect ϵ_f when applied will elevate the effect

$$[\text{T-PureApp}]\Gamma \vdash e_1 : \tau_1 \xrightarrow{\mathbf{Pure}} \tau_2[\mathbf{Pure}] \quad \Gamma \vdash e_2 : \tau_1[\mathbf{Pure}] \quad \Gamma \vdash e_1\, e_2 : \tau_2[\mathbf{Pure}]$$

$$[\text{T-IOApp}]\Gamma \vdash e_1 : \tau_1 \xrightarrow{\mathbf{IO}} \tau_2[\epsilon_1] \quad \Gamma \vdash e_2 : \tau_1[\epsilon_2] \quad \Gamma \vdash e_1\, e_2 : \tau_2[\mathbf{IO}]$$

$$[\text{T-Lam}]\Gamma, x : \tau_1 \vdash e : \tau_2[\epsilon] \quad \Gamma \vdash \lambda(x : \tau_1).\, e : \tau_1 \xrightarrow{\epsilon} \tau_2[\mathbf{Pure}]$$

$$[\text{T-Print}]\ \Gamma \vdash \mathbf{print}(e) : \mathbf{unit}[\mathbf{IO}]$$

$$[\text{T-SubEffect}]\epsilon_1 \preceq \epsilon_2 \quad \Gamma \vdash e : \tau[\epsilon_1] \quad \Gamma \vdash e : \tau[\epsilon_2]$$

$$[\text{T-Cost}]\Gamma \vdash e : \tau[\epsilon] \quad \text{CostInfer}(e) \leq C \quad \Gamma \vdash \mathbf{cost}\{e\} : \tau[\epsilon]$$

$$[\text{T-Proof}]\Gamma \vdash e_{prf} : P\ \mathbf{true}[\mathbf{Pure}] \quad \Gamma \vdash \mathbf{proof}\{e_{prf}\} : \mathbf{Proof}[\mathbf{Pure}]$$

Figure 2: Selected typing rules for gp-λ. T-PureApp and T-IOApp show function application in pure vs. effectful cases. T-SubEffect allows an expression to be used in a larger effect context ($\epsilon_1 \preceq \epsilon_2$ means effect ϵ_1 is no greater than ϵ_2 in the lattice, e.g. Pure $\preceq$ IO). T-Cost checks a cost block via a static cost inference CostInfer. T-Proof checks a proof block given a proof goal P (here simplified as P **true** meaning P is proven).

of the context to ϵ_f. We enforce a simple subeffecting: an expression with a smaller effect can be used in a bigger effect context (rule T-SubEffect). For example, a pure integer can be used wherever an **IO** integer is needed (with no runtime impact), as Pure $\preceq$ IO.

Side-effect operations. We have included a rule T-Print as representative of an I/O primitive: printing to console yields type **unit** and is an **IO** effect. Similarly, one could have rules for file or network operations yielding **FS** or **Net** effects. These rules simply introduce the appropriate effect in the typing judgment.

Cost blocks. The rule T-Cost is crucial for resource tracking. CostInfer$(e) \leq C$ denotes the result of a static cost inference for e is bounded by C. We do not fully elaborate the inference algorithm here; it can be thought of as a form of abstract interpretation or typing derivation that computes a numeric or symbolic cost for e based on its structure (e.g., a loop from 1 to n might infer cost $O(n)$). This is where we leverage AARA techniques: each operation in e might be assigned a potential, and C is checked against the sum of potentials consumed:contentReference[oaicite:27]index=27:contentReference[oaicite:28]index=28. If e's inferred cost is greater than the annotation C, the type checker will reject the program (meaning the model either mis-annotated the cost or

wrote a too-slow implementation). If the cost check succeeds, **cost**$\{e\}$ has the same type as e and the effect of e (since the cost wrapper itself doesn't introduce new side effects). For example, if e is a pure loop that runs in linear time $O(n)$ and $C = O(n)$, the rule allows it; if C was claimed as constant but inference says $O(n)$, it fails.

Notably, the cost checking relies on a sound but perhaps incomplete static analysis. The philosophy is that an AI generating code can also generate a corresponding cost bound; if it guesses incorrectly, the type checker (or verifier) informs it, and it can adjust. Over time, one could imagine the model learning to make correct cost annotations, analogous to how humans learn complexity analysis. The formal guarantee (proven in Section 3) is that if $\Gamma \vdash \mathbf{cost}\{e\} : \tau$, then indeed e will run within the specified cost at runtime (cost soundness theorem).

Proof blocks. The rule T-Proof is a bit abstract in our presentation. Essentially, if there is a proposition P that needs to be proved at a program point, we expect the proof block to contain a proof term e_{prf} that establishes P. We denote this by saying e_{prf} has type P **true**, read as "P is true". In practice, P could be any logical formula about program variables or functions (e.g., a sortedness invariant or a functional correctness specification). The rule says that if such a proof term is well-formed (checked by a proof checker, possibly an SMT solver or interactive theorem prover), then the **proof**$\{e_{prf}\}$ expression can be typed as **Proof** (a placeholder type signifying a discharged obligation). The effect is **Pure** because writing a proof has no computational side effects.

In a typical usage, a proof block would appear at a position in code where a certain assumption needs justification. For instance, after a loop, one might need to prove that a variable indeed holds the sum of the first n numbers. The gp-λ code could include:

$$\mathbf{proof}\{\ \textit{(proof that } s = 1 + 2 + \cdots + n\textit{)}\ \}\,,$$

which the type checker will verify using logical reasoning. If the proof is not correct or incomplete, the program is not accepted. This mechanism ensures that the model cannot simply assert a property without actually providing evidence—addressing the hallucination of correctness. It forces a kind of *explainability*: the model must "show its work" for critical claims.

To keep the presentation manageable, we have omitted some standard rules (variables, let-binding, etc.) which are straightforward. Conditionals and loops would follow standard typing: the guard of a conditional must be

Bool; a loop '**while** $e_1\{e_2\}$' requires e_1 : **Bool** and that e_2 has type **unit** (with some invariant proof obligation likely accompanying it, omitted here). Memory and mutation (var, assignment) can be handled via a state effect or a Hoare logic embedded in proofs, but a full treatment is outside our scope—our focus is on the novel aspects of cost and proof in a model-centric setting.

2.3 Operational Semantics and Resource Cost

We give a brief overview of gp-λ's operational semantics, focusing on how cost is accounted for. The semantics is defined as a small-step relation $e \xrightarrow{t} e'$ where t is a non-negative integer cost tick. A step can either be a computational step (with $t = 1$ for a basic β-reduction or primitive operation) or an internal logical step (such as reducing inside a proof, which we might treat as $t = 0$ since proofs don't count toward runtime cost).

For example, the β-reduction rule for a pure function is:

$$(\lambda(x : \tau).\, e)\ v \xrightarrow{1} e[x := v]\,,$$

which consumes 1 unit of cost. A primitive operation like addition $n_1 + n_2$ might have:

$$n_1 + n_2 \xrightarrow{1} n_{\text{sum}}\,,$$

again counting as 1 step. A print statement could be:

$$\mathbf{print}(v) \xrightarrow{1} ()\,,$$

with an observable I/O effect (we assume a model of the external world but skip details).

A cost block $\mathbf{cost}\{e\}$ does not itself create a new runtime behavior beyond e; it's more of a static annotation. At runtime, one can assume $\mathbf{cost}\{e\}$ simply behaves as e (with maybe a no-op wrapper). However, for the cost semantics, we enforce that if $\mathbf{cost}^C\{e\}$ is in the program, the total cost to evaluate e (to a value) must not exceed C. We could formalize this by an evaluation relation that accumulates cost: e.g., $\langle e, 0\rangle \to^* \langle v, k\rangle$ meaning e evaluates to value v with k cost. Then a soundness condition would be: if $\vdash \mathbf{cost}^C\{e\} : \tau$ and e evaluates to v in k steps, then $k \le C$ (see Section 3 for the proof). In a monitored runtime, one might even instrument cost blocks to abort if exceeded, but here we rely on static guarantees.

Proof blocks $\mathbf{proof}\{e_{prf}\}$ are erased in the actual operational semantics. That is, $\mathbf{proof}\{e_{prf}\}$ evaluates in one step to () (unit) or just disappears,

with e_{prf} not affecting the runtime state (aside from maybe an off-line record that a proof was checked). We do not charge cost for proof execution in the operational cost count, under the assumption that proof checking is done at compile-time (or by a separate process). This separation is important: we don't want the model to avoid using proofs for fear of runtime cost—proofs are free in terms of the program's execution, though they have "verification cost" outside of it.

In summary, gp-λ's operational semantics is standard for a call-by-value functional language with side effects. The unique aspect is the introduction of a cost metric, which is orthogonal to the standard semantics, and the treatment of proofs as ghost code. We ensure in our Coq model that neither the presence of cost annotations nor proofs can interfere with the usual execution (preservation theorem takes those into account, essentially treating them as annotations).

3 Mechanized Semantics in Coq

We have developed a mechanized specification of gp-λ in the Coq proof assistant. In this section, we outline the mechanization and highlight the main theorems and their proofs (at a high level). Mechanizing the language serves to increase our confidence in gp-λ's soundness and provides a platform for future extensions (e.g., mechanized proof search or certified code generation).

3.1 Coq Formalization of Syntax and Typing

In Coq, we define inductive types for the abstract syntax of gp-λ. For instance:

```
Inductive type : Type :=
  | TInt : type | TBool : type | TString : type
  | TUnit : type
  | TArrow : effect -> type -> type -> type
  | TProofObj : type
with effect : Type := Pure | IO | FS | Net.
Inductive expr : Type :=
  | EVar : var -> expr
  | EConst : const -> expr
  | ELam : var -> type -> expr -> expr
  | EApp : expr -> expr -> expr
```

```
| EIf : expr -> expr -> expr -> expr
| ELet : var -> expr -> expr -> expr
| ECost : costAnn -> expr -> expr
| EProof : proposition -> proof_term -> expr
| EUnit : expr
| EVarDecl : var -> expr -> expr -> expr
| EAssign : var -> expr -> expr
| EWhile : expr -> expr -> expr
| EPrint : expr -> expr.
```

Here costAnn represents a cost annotation (e.g., a numeric bound or a polynomial expression in some variables), and proposition and proof_term represent the logical claim and its proof (which we keep abstract, as Coq can treat them as opaque or interact with an SMT solver for checking). This deep embedding of syntax allows us to reason about programs as mathematical objects in Coq.

The typing relation is encoded as an inductive predicate has_type Gamma e ty eff, corresponding to $\Gamma \vdash e : ty[eff]$. We have one constructor for each typing rule (like T_Lam, T_AppPure, T_Cost, etc.). For example, the cost rule appears as:

```
| T_Cost : forall Gamma e ty eff C,
    has_type Gamma e ty eff ->
    cost_infer e <= C ->
    has_type Gamma (ECost C e) ty eff
```

This directly mirrors the math rule, with cost_infer e <= C being a Coq function that implements the static cost analysis (we proved this function correct separately).

Likewise, the proof rule might look like:

```
| T_Proof : forall Gamma P pf,
    (* check that pf proves proposition P: *)
    proves Gamma pf P ->
    has_type Gamma (EProof P pf) TProofObj Pure
```

where proves Gamma pf P is a relation connecting a proof term to a proposition in context.

We took care to ensure the type system is syntax-directed and decidable, so that a type-checker can be implemented (in fact, we extracted a prototype type-checker from the Coq definitions to test it on example programs).

3.2 Operational Semantics and Cost Accounting

The small-step operational semantics is defined as an inductive relation step : state -> state -> nat -> Prop, where the relation carries a cost (nat) for the step. A state encapsulates the expression being evaluated and any environment for I/O or mutable state (for simplicity, our Coq model treats I/O abstractly and uses a partial function for variable store). A typical step rule in Coq looks like:

```
| S_Beta : forall x tau eff body v,
    value v ->
    step (ELam x tau body, env) (subst x v body, env) 1
```

This corresponds to the β-reduction with cost 1. We also have rules for each construct. The cost and proof constructs are handled as:

```
| S_Cost : forall C e v env,
    value v ->
    eval (e, env) v -> (* if e fully evaluates to v, no step cost here *)
    step (ECost C e, env) (v, env) 0
| S_Proof : forall P pf env,
    step (EProof P pf, env) (EUnit, env) 0.
```

Here, eval is a multi-step evaluation (we use it to reason about terminating runs). The cost block, when it finishes evaluating e to v, transitions to v with no additional step cost (the cost was accounted internally during the eval of e). The proof block goes to unit immediately with 0 cost. Essentially, cost and proof are no-ops at runtime (all the action was at compile-time).

3.3 Metatheory: Preservation, Progress, and Cost Soundness

Using the Coq development, we prove the following main theorems:

Theorem 1 (Type Soundness). *(Preservation)* If $\Gamma \vdash e : \tau[\epsilon]$ and $e \xrightarrow{t} e'$ (in one step), then there exists ϵ' such that $\Gamma \vdash e' : \tau[\epsilon']$ and $\epsilon' \preceq \epsilon$. Furthermore, the type τ remains the same. (In other words, a single step of evaluation preserves types and does not increase the effect level.)

(Progress) If $\emptyset \vdash e : \tau[\epsilon]$, then either e is a value (a fully evaluated normal form), or e can take a step $e \xrightarrow{t} e'$ for some e' and cost t. Moreover, if $\epsilon = \textbf{Pure}$, then e cannot be a stuck state waiting on an I/O operation (pure computations cannot get stuck on effects).

These are proven by the usual inductive techniques on the derivations, with some twists due to cost and effects. The presence of the effect in the typing judgment required carefully strengthening the induction hypotheses to show that even if an expression steps to a higher-effect expression, we have the subeffect rule to keep type preservation valid. For example, printing has no next step (it's a value after printing unit), so progress requires that an **IO** expression can either step or is a value of type **unit** (which in our semantics means the I/O action has been performed). We formalize this using a small-step with states that include an event for I/O. The Coq proofs closely follow Wright and Felleisen's approach to type safety:contentReference[oaicite:29]index=29, adapted to a setting with effects (for which we rely on a progress lemma that pure parts progress normally and effectful parts either perform an effect or progress).

Theorem 2 (Cost Soundness). *If* $\emptyset \vdash \mathbf{cost}^C\{e\} : \tau[\epsilon]$ *and* e *evaluates to a value* v *in* k *steps (i.e.,* $\langle e, 0\rangle \to^* \langle v, k\rangle$*), then* $k \leq C$*.*

This theorem states that any well-typed cost block indeed respects its declared resource bound. The proof in Coq proceeds by induction on the typing derivation of $\mathbf{cost}\{e\}$. By rule T-Cost, we know that $\text{CostInfer}(e) \leq C$ and that $\Gamma \vdash e : \tau[\epsilon]$. We then use a strengthened invariant: we proved a lemma that *if* $\Gamma \vdash e : \tau[\epsilon]$ *and* $\text{CostInfer}(e) = B$*, then for any evaluation of* e *to* v *with cost* k*, we have* $k \leq B$*.* This lemma itself is proven by structural induction on e, leveraging the AARA-style potential annotations. For example, the inductive step for a loop uses the potential assigned to the loop to cover an iteration plus the potential for the remaining iterations, thereby proving a bound. Combining that lemma with $\text{CostInfer}(e) \leq C$, we get $k \leq C$ as required.

This cost soundness property is crucial in bridging the static and dynamic aspects of gp-λ. It provides a machine-checked guarantee that any program accepted by the gp-λ type checker will run within the resources it declares. In practical terms, if an AI writes a gp-λ program with a `cost{}` annotation, we can trust (by Theorem 2) that the program will not exceed that cost at runtime, eliminating a whole class of performance bugs and denial-of-service vulnerabilities.

Theorem 3 (Proof Soundness). *If* $\emptyset \vdash \mathbf{proof}\{e_{prf}\} : \mathbf{Proof}[\mathbf{Pure}]$*, then the proposition* P *associated with* e_{prf} *is logically true (under the interpretations of any assumptions in* Γ*).*

This theorem connects the typing of proof blocks with the correctness

of the proved properties. In simpler terms, it says our type system does not accept bogus proofs: any proof term that type-checks indeed proves the intended property. The proof of this relies on the soundness of the external proof-checker or logic we assumed for proves. We assume if proves Gamma pf P holds (which was required by T-Proof), then P is semantically true. This assumption can be discharged if we embed an actual logic (e.g., we tie proves to Coq's internal logic, then this theorem is essentially stating Coq's consistency, which we take as given:contentReference[oaicite:30]index=30). Thus Theorem 3 is more about trusting the meta-theory of our proof system rather than a new property of gp-λ per se.

Nevertheless, Theorem 3 ensures that gp-λ lives up to its verification intent: any property the program claims and provides a proof for is indeed a valid property of the program's behavior. For example, if a gp-λ program has a proof block claiming that a sorted list remains sorted after an insertion sort, and the block type-checks, we know the sort function is partially correct (assuming termination, which could itself be a property requiring proof).

All these theorems were mechanized. The development consists of about 5,000 lines of Coq code (including definitions and proofs). We benefited from prior formalization techniques: progress and preservation follow the standard textbook approach (with slight extensions for effects), and cost soundness was proved using a technique similar to the one used by Hoffmann and colleagues for AARA soundness proofs:contentReference[oaicite:31]index=31 (they often prove that a typing derivation implies an evaluation bound). One noteworthy aspect is that our cost analysis in Coq had to be carefully restricted to remain decidable and relatively complete for typical patterns; we ended up implementing a potential-based analysis that can handle linear and some polynomial costs, which was sufficient for our examples.

3.4 Integration with AARA-style Resource Analysis

It is worth elaborating how our approach ties in with AARA, as this might be of independent interest. In AARA, one assigns *potential* to data structures such that the consumption of potential as the program runs yields a bound on resource usage:contentReference[oaicite:32]index=32. We mimicked this in gp-λ's type system by introducing latent cost coefficients in typing rules (not visible in the simplified rules earlier). For instance, the typing rule for a loop **while** in our actual Coq development requires a certain amount of potential to be available to account for each iteration. We then prove a *soundness of cost inference*: if the type system says a program can be typed with a cost bound C, then indeed C is an upper bound on the cost. This is

conceptually analogous to Hofmann and Jost's soundness theorem for their type system:contentReference[oaicite:33]index=33.

However, gp-λ is not *automatic* in cost inference: the model or programmer is expected to supply the bound C. This is a design choice to favor simplicity of the type system (no complex refinements or linear constraints are part of the types) at the cost of requiring the generator to guess the bound. We expect an AI coder could attempt a few different C forms or use a heuristic to get it right. The benefit is that the type checking (verifying a given C) is then a straightforward numeric check, not requiring solving linear programs or SMT queries during compilation. This aligns with a model-centric design: push the hard task (coming up with an invariant or bound) to the AI, and keep the verifier simple and fast (for real-time feedback).

In summary, our mechanized semantics and proofs confirm that gp-λ is internally consistent and achieves the intended safety properties. With these foundations, we move on to using the language in practice.

4 Example Programs in gp-λ

We now present several example programs written in gp-λ, showcasing how its features can be used to write and verify code. For brevity, we use a pseudo-code style that is close to gp-λ's concrete syntax (which the model would actually produce). All examples have been type-checked and verified in our implementation.

4.1 Pure Recursion: Factorial

Our first example is a simple pure function: computing the factorial of a number. This illustrates gp-λ's basic λ-calculus core for total functional programming.

Listing 1: Pure recursive function (factorial) in gp-λ.

```
fun fact(n: Int) : Int =
  if n <= 1 then 1 else n * fact(n-1)
```

In Listing 1, `fact` is defined as a recursive function (using an implicit recursion through self-reference; one could also use a `fix` combinator, but here we assume top-level `fun` defines a potentially recursive function as is common in ML-style languages). The function's type is `Int` $\rightarrow$ `Int`, inferred automatically: it is **Pure** because it performs no I/O, and indeed gp-λ will verify that `fact` only calls itself and uses arithmetic.

Totality (termination) is an interesting point: as written, `fact` will loop indefinitely if called with a negative number. In gp-λ, we could prevent this by refining the type of `n` to a natural number (non-negative Int) and then proving that the recursion decreases. Alternatively, we might leave it as Int but then we cannot guarantee termination for negative inputs. Our effect system could classify potentially non-terminating computations as a separate effect (e.g. **Div** for divergence) as done in some languages:contentReference[oaicite:34]index=34. For simplicity, we assume well-formed use or that an additional proof obligation (not shown) exists to show termination. The key point for our example is that no side effects or cost annotations are involved, so `fact` is a straightforward piece of code that an LLM could generate easily, and gp-λ accepts it as is.

4.2 I/O with Effect Tracking: Greeting Example

Next, we illustrate an I/O effect and how the type system tracks it. Consider a program that greets a user given their name:

Listing 2: An I/O example with effect tracking.

```
fun greet(name: String) : unit =
  print("Hello, " + name + "!")
```

In Listing 2, the function `greet` concatenates a greeting and prints it. The type of `greet` is `String` $\rightarrow$ `unit`. Gp-λ infers or checks the **IO** effect because the `print` operation is of effect **IO** (rule T-Print in Figure 2). If we erroneously declared `greet` as `String` $\rightarrow$ `unit`, the type checker would reject it, because a **Pure** function is not allowed to perform printing. This enforcement is important for an AI model: if it attempts to perform I/O in a context that is supposed to be pure (say, calling an API that expects a pure callback), gp-λ's type system will immediately flag the issue. In other words, effect mismatches (which might be subtle bugs in a traditional language) become explicit type errors that the model must correct (likely by adjusting the types or restructuring the code).

For comparison, in languages like Haskell, the type of `greet` would be `String -> IO ()`, and using it in a pure context is a compile-time error. Gp-λ offers a similar safety net, but in a form readily checkable by an automated theorem prover or even another LLM acting as a verifier. This helps address the "illusion" issue mentioned earlier:contentReference[oaicite:35]index=35: the model cannot accidentally treat an effectful operation as pure knowledge, it must respect the separation which ultimately reduces inconsistent or impossible completions.

4.3 Cost-Verified Loop: Summation

Now we demonstrate gp-λ's cost verification in action. Consider a function that computes the sum of numbers from 1 to n. We want not only to implement this, but also to verify that it runs in $O(n)$ time. We use a cost block to enforce a linear bound.

Listing 3: Loop with cost annotation ensuring linear time complexity.

```
fun sum_to(n: Nat) : Int =
  cost { // cost  O(n)
    var i = 0;
    var s = 0;
    while i < n {
      s := s + i;
      i := i + 1;
    }
    return s
  }
```

In Listing 3, `sum_to` is defined with a cost block around the loop. The comment `// cost  O(n)` is an informal note; formally, we would attach a cost metric, say $C = an+b$ for some constants. The type of `sum_to` is `Nat` $\rightarrow$ `Int` (we use `Nat` to denote a non-negative Int type, to avoid concerns about loop termination on negative values). The entire function is pure from the perspective of side-effects (it just does arithmetic, no I/O), but it has an important resource effect: it runs in linear time.

The cost block tells the type checker: "the following code should run in linear time." The checker then uses the cost inference to verify this. In the loop: - It sees a loop that iterates n times (from $i = 0$ to $i < n$). - The body of the loop does constant work (two additions and an assignment). - So it infers cost $\Theta(n)$ for the loop. If the bound $O(n)$ is interpreted as say $C = c*n+d$ for some c, d, the inference CostInfer(e) would yield something like $1*n+0$ (with potential method yielding 1 unit per iteration), which is indeed $\leq c*n+d$ for appropriate choice. Thus the program type-checks.

What if the model wrote something less efficient? For example, summing numbers by nested loops (making it $O(n^2)$) but still annotated it as $O(n)$ cost. Then CostInfer(e) might yield $\sim n^2$, which would not be $\leq C$ if C was linear, and the type checker would reject it. The model would have to either correct the algorithm or adjust the annotation to $O(n^2)$. If it adjusts to $O(n^2)$, the program would type-check but now carries a clear indication of worse complexity—perhaps leading a higher-level system to flag it as suboptimal or ask for improvement. In this way, gp-λ not only verifies

complexity but could guide the model towards more efficient solutions.

The explicit `cost{}` block in the code is an example of gp-λ's transparency: rather than hiding complexity in big-O comments or leaving it implicit, the code itself states the expected complexity, and that statement is checked. This is particularly useful for AI-generated code, as it ensures the model's "intent" (as expressed by the annotation) matches the code's reality, or it gets immediate feedback to reconcile the two.

4.4 Static Resource Bound Enforcement: Memory Allocation Example

For a final example, we illustrate how gp-λ can enforce a memory (or more generally, resource) bound. Suppose we have a system with limited memory and we want to prevent a function from allocating more than a certain amount. We can integrate that into the cost model as well (treat memory allocation as a resource cost).

Consider a function that creates an array of size n. We want to ensure it does not allocate more than n elements, obviously. If the model accidentally uses a double loop to fill it (thus allocating n^2 in total erroneously), the cost system should catch it.

For simplicity, we illustrate with pseudo-array code (assuming we had arrays and an allocation cost counted):

Listing 4: Static enforcement of allocation bound using cost blocks.

```
fun init_array(n: Nat, val: Int) : Array(Int) =
  cost { // cost  O(n) space
    var arr = new Array<Int>(n);
    var i = 0;
    while i < n {
      arr[i] := val;
      i := i + 1;
    }
    return arr
  }
```

In Listing 4, `init_array` allocates a new array of length n and initializes all entries to `val`. We annotate it with a cost block expecting $O(n)$ cost (here cost could count both time and space in the same big-O, or we might have separate annotations; let's assume space is included). The type of `init_array` is `Nat` $\times$ `Int` $\rightarrow$ `Array(Int)` since it allocates memory (which we classify as a filesystem effect or a more general resource effect). The cost inference sees: - The allocation `new Array<Int>(n)` costs n (space for

n entries). - The loop costs n steps to initialize. Total cost $\approx 2n$, which is $O(n)$, so it fits the annotation.

If the model had mistakenly written a double loop (like allocate n, then append n elements one by one doubling the allocation, etc.), the inference might see something like $n + n^2$ and not accept $O(n)$ as an upper bound. This forces correction.

This example shows gp-λ can be used to enforce not just time complexity but other resource constraints (e.g., memory, network calls, etc.) by extending the cost model. It's up to the specification what "cost" measures; our framework in Coq is general enough to accommodate multiple resources by vector of potentials (though we focused on a single metric in theory). The big picture is that any resource that an LLM might inadvertently overuse can be guarded by a cost annotation, turning a potential runtime problem into a compile-time (model-time) feedback.

Summary of Examples

Through these examples, we've seen that: - Gp-λ can express ordinary algorithms in a style close to mainstream imperative or functional pseudocode (so an AI trained on typical code can adapt to it easily). - The additional annotations (`cost` and `proof`) and effect labels do impose some overhead, but they are quite lightweight compared to full formal verification in languages like Coq. For instance, our cost annotations did not require specifying the exact function, just the complexity class $O(n)$. The proof in the summation example was trivial (we didn't explicitly show it, but it could be as simple as an inductive argument that the loop adds correctly—likely a few lines of logical steps). - Each example that had an annotation or special block received a guarantee: the factorial had none (except an implicit termination expectation), so it only got type safety; the I/O example's guarantee is that no hidden effects occur; the summation got a complexity guarantee; the array initialization got a space guarantee. In a normal language, none of these properties would be checked by the compiler. In gp-λ, they are part of the contract of the code.

From a user (model) perspective, writing these annotations is part of the coding process. It might seem burdensome to a human, but for an AI, it's just another prediction task. We envision an AI-assisted coding workflow where the model proposes code along with these annotations by default. Because gp-λ is designed to be amenable to automated reasoning, the model can be trained or engineered (via prompt or system tools) to supply correct or at least checkable annotations most of the time.

With concrete programs in hand, we now compare gp-λ to existing languages to clarify its niche and learn from prior art.

5 Comparison to Related Languages

Gp-λ intersects with ideas from several domains: refinement type systems, dependent type theory, effect systems, and formal verification of program properties. We compare and contrast gp-λ with three prominent language-based approaches, focusing on (i) syntax compactness, (ii) verification power, and (iii) model-friendliness of each.

5.1 Liquid Haskell

Liquid Haskell (LH) is an extension of Haskell that incorporates refinement types to allow lightweight formal verification on top of Haskell's typing:contentReference[oaicite:36]index=36. For example, one can specify that a list is non-empty by refining its type, or that an index is within bounds, and LH uses an SMT solver to automatically prove many of these properties. Liquid Haskell's design philosophy is to stay close to Haskell's natural syntax and require minimal manual proofs—properties are enforced by implication checking of refinements, and when proofs are needed, they can often be just a few lines (like a lemma that gets automatically checked).

In terms of **verification power**, Liquid Haskell is quite powerful for safety properties and even termination checking (it proved 96% of recursive functions terminating in one evaluation, mostly automatically:contentReference[oaicite:37]index=37). However, it deliberately limits expressiveness to remain automatic: it doesn't allow full arbitrary proofs of complex theorems within the code; instead, it restricts to properties expressible as refinements (essentially predicates over values, often in the quantifier-free logic that the SMT can handle). In contrast, gp-λ allows arbitrary propositions to be proved inside `proof{}` blocks, essentially offering a pathway to full verification of functional correctness or other complex properties (if the model can provide the proof). This aligns gp-λ more with interactive theorem provers or languages like F* in terms of verification scope, whereas Liquid Haskell is aimed at automating a narrower class of properties.

On **syntax compactness**, Liquid Haskell inherits all of Haskell's syntax (which is already quite compact compared to, say, Java, but still includes things like do-notation, where clauses, etc.). Refinement annotations in LH are usually provided in comments or as special annotations (e.g., `{-@ ... @-}` syntax for specification). This means a source file contains both Haskell

code and extra specification text. An LLM generating Liquid Haskell would need to manage both, which can be verbose, though not overwhelmingly so. Gp-λ, by design, integrates specification as code (cost and proof blocks) and keeps the overall syntax minimal. For instance, verifying a simple property in Liquid Haskell might require writing a separate lemma function with a refinement type and calling it, whereas gp-λ could just use a `proof{}` inline. That said, one advantage LH has is that the majority of proofs are discharged by the SMT solver automatically, whereas gp-λ might require the model to explicitly construct proofs. There is a trade-off: LH aims for automation at the cost of needing an external solver and restricting what can be verified; gp-λ aims for transparency and explicitness, which could result in more verbose proofs but ones that a model can understand as just another piece of code.

For **model-friendliness**, we consider how each fares for an AI code generator. Liquid Haskell's human-oriented design (leveraging Haskell idioms and heuristics for the SMT) can be a double-edged sword. On one hand, if an LLM is fine-tuned on a lot of Haskell code, generating Haskell syntax is fine—but generating correct refinements might be tricky without understanding the property. On the other hand, gp-λ requires the model to output proofs, which might seem harder. However, gp-λ offers a simpler, more uniform playing field: there are fewer implicit things (everything to be checked is right there in the code for the model to consider). Also, gp-λ doesn't have features like type classes, higher-kinded types, etc., which Haskell does—these can confuse an LLM or lead to more complex interactions. By avoiding such features, gp-λ might actually be easier for an AI to reliably generate in the domain of verification tasks.

In summary, Liquid Haskell provides a great deal of automated checking with minimal annotation (good for human convenience), whereas gp-λ demands more explicit annotations (cost, proof) but in return provides stronger guarantees and a more straightforward verification model. Gp-λ trades some of LH's automation for flexibility and explicitness, anticipating that the burden of writing those extra annotations is taken on by an AI, not a human.

5.2 RustBelt Core (Rust)

Rust is a systems programming language focusing on memory safety and performance. Its core type system (ownership with lifetimes and borrowing) ensures that, in safe Rust code, there are no use-after-free, no data races, and other common bugs. The RustBelt project formalized a large subset

of Rust, defining a core calculus (often called λ_{Rust} informally) and proving its safety in Coq:contentReference[oaicite:38]index=38. RustBelt's success demonstrates that advanced type systems can provide strong guarantees without full specification of program intent: Rust's guarantees are mostly around memory and thread safety, not general functional correctness.

In terms of **verification power**, gp-λ and Rust aim at different properties. Rust's type system cannot, for example, prove that a sort function actually sorts, or that an algorithm runs within a certain time—it focuses on low-level safety. Gp-λ, on the other hand, is built to verify high-level properties (correctness, complexity) but does not inherently enforce memory safety (we rely on proofs or the discipline of the AI to avoid unsafe behaviors like out-of-bounds access, unless those are specified as proof obligations). It would be possible to add Rust-like ownership to gp-λ (perhaps as part of the effect system or a linear type qualifier) to prevent memory misuse, but currently gp-λ assumes a managed memory model or that such issues are handled by proofs if needed.

On **syntax**, Rust is quite verbose compared to gp-λ (with curly braces, keywords, etc.), though not excessively so (and it has type inference to reduce annotation burden for humans). For an LLM, Rust's vocabulary and rules (like lifetimes) could pose a challenge; indeed, today's code models sometimes struggle with Rust's borrow checker-related code, often requiring several attempts to satisfy it. Gp-λ has a simpler core syntax and fewer keywords. In the context of a top AI systems conference, one might ask: if an AI can generate safe Rust, why not just use Rust? The reason is that Rust's rules are very specific (memory safety) and its enforcement is mostly automatic but limited to that domain. Gp-λ asks the AI to do more work (explicit proofs, cost annotations) but in return, we get a language that can verify things Rust never tries to, and is more "open-ended" in specifiable guarantees.

Concerning **model-friendliness**, Rust was designed for humans, and although it has a stricter discipline than something like Python, it still contains complexities (like distinctions between stack and heap, mutability, etc.) that an AI might not always navigate correctly. We have seen that large models sometimes generate Rust code that fails to compile due to borrow checker issues or other subtle mistakes:contentReference[oaicite:39]index=39. In gp-λ, the analogous situation would be failing to type-check due to a proof or cost issue. The crucial difference is that gp-λ provides a way for the model to gradually fix the issue by adding proof steps or adjusting bounds, whereas in Rust, if the borrow checker rejects code, the model must implicitly figure out a different approach or rephrase the code—something even human

novices find hard. At least gp-λ pinpoints the unmet obligation (e.g., "you claimed $O(n)$ but I infer $O(n^2)$"), which is a clearer signal for an AI to act on than Rust's often cryptic lifetime errors.

RustBelt's formalization:contentReference[oaicite:40]index=40is relevant insofar as it shows how to rigorously prove a real-world type system safe. We have taken inspiration in mechanizing gp-λ in Coq similarly. One key difference: Rust's safety was proved once and for all by experts and is hard-wired; gp-λ shifts some proving to the user (or model) in the form of proof blocks for arbitrary properties. In a sense, gp-λ is less opinionated—it provides a framework for verifying whatever you need (including memory safety if you formalize that), rather than building one specific safety guarantee in. This generality makes it more flexible but also more dependent on the AI's capabilities.

5.3 F*

F* is a dependently-typed programming language and proof system in which programs with effects can be verified against specifications:contentReference[oaicite:41]index=41. It can be seen as an ML-like language that, like Coq or Agda, allows the expression of rich specifications, but unlike those, it integrates an effect system and leverages SMT solvers to automate many proof obligations. F* has been used to verify complex cryptographic libraries and protocols (e.g., TLS implementations) by encoding properties as dependent types or using its Hoare logic inspired frameworks:contentReference[oaicite:42]index=42.

In terms of **verification power**, F* is one of the most powerful systems available; one can prove full functional correctness, security properties, and even meta-theorems within F*. Gp-λ aspires to similar power (we allow arbitrary proofs), but F* has a huge head start in terms of existing libraries, automation (like built-in tactics, SMT integration), and expressiveness (full dependent types, user-defined effects, etc.). For example, F* can define a user effect for nondeterminism or distributed state and prove things about it; gp-λ currently has a fixed small set of effects. However, with more development, gp-λ could emulate many F* features by encoding them via proofs and effects.

Where gp-λ shines is **syntax compactness**. F*'s syntax, while inspired by ML, ends up being fairly heavy in practice, especially when doing proofs. One often has to write ghost code, explicit type qualifiers, and lemmas. The F* code for even simple things can become verbose (though not as much as Coq, thanks to SMT automation). Gp-λ tries to keep the surface syntax simple: our effect annotations are a single keyword on arrows (like in listing

examples), cost and proof are explicit blocks (bounded by braces), which are not much different than writing an assert or a lambda. We intentionally avoided introducing a full dependent type language syntax (with Π types or where clauses, etc.) — instead, we offload specifications either to refined types (which we could add in a Liquid Haskell style if needed) or to assertions proven in proof blocks. This keeps the core language context-free and easier to generate. An LLM might prefer gp-λ to F* because in gp-λ, every needed token is more predictable (there's less type inference magic or universe polymorphism or other advanced concept to juggle).

In terms of **model-friendliness**, F* poses a challenge for current LLMs. Writing correct F* requires understanding of advanced type theory and the underlying proofs. If a model is not specifically trained or engineered to produce such proofs, it will struggle. Gp-λ doesn't eliminate that difficulty (the model still has to prove things), but it confines it into proof blocks that could possibly be tackled by specialized tools or by iteratively querying a theorem-prover. A realistic scenario: an LLM generates a gp-λ program with some holes in proofs, then a proof assistant (maybe another AI or tactic system) fills them in, or the LLM itself tries until it passes. This decomposition (code vs proof vs resource) is arguably clearer in gp-λ. In F*, the code and proofs are often interwoven, which might confuse an AI (commonly mixing computations with logical refinements).

One area F* doesn't emphasize is **cost analysis**. While one can do it manually in F* (by adding a fuel count or using its effect to count steps), it doesn't have a built-in notion of cost checking. Gp-λ explicitly includes that, which is a point in its favor for certain applications (embedded or real-time systems, for instance, where knowing time complexity is crucial, or smart contracts where gas usage needs bounding:contentReference[oaicite:43]index=43). By citing AARA:contentReference[oaicite:44]index=44, we grounded our approach in known research, but made it part of the language surface (so the model is aware of it). F* could encode AARA but it's not currently a native feature.

In summary, F* represents the upper end of verification capability and gp-λ is a more focused, perhaps simpler system that doesn't reach as far (yet) in terms of proof automation or features, but is intentionally designed for an AI usage model. A fruitful comparison is that gp-λ might achieve 80% of what F* can verify, but with code that's 50% shorter. If that trade-off holds, it's a win for machine generation, where brevity translates to less chance of error and fewer tokens to model.

Summary of Comparisons

Table 1 summarizes the comparison:

Table 1: Qualitative comparison of gp-λ with Liquid Haskell, Rust (core), and F*.

Language	**Verification scope**	**Syntax / annotations**	**Optimized for AI?**
gp-λ	Functional correctness (with manual proofs), complexity bounds, effect safety. General-purpose.	Minimalistic functional syntax; explicit `cost` and `proof` blocks; simple effect labels. Annotations are part of syntax.	Yes: designed for compactness and explicit machine-checkable structure (few hidden rules).
Liquid Haskell	Partial correctness (refinement properties), termination. Automated via SMT (no explicit proofs).	Haskell syntax + refinement specs in comments. Little extra annotation for user (SMT does heavy lifting).	No: designed for humans. LLM can use it, but must invoke external solver to discharge proofs.
Rust (RustBelt core)	Memory safety, data race freedom (via ownership). No high-level spec verification.	C/OCaml-like syntax with explicit mutability and lifetimes (sometimes verbose). Safety mostly automatic (inferred).	No: human-oriented; very strict rules that are non-negotiable (could frustrate model, though they prevent certain errors).
F*	Full verification (dependent types allow any property); user-defined effects; SMT + manual proofs.	ML-like syntax, but heavy with type annotations, lemmas, and sometimes verbose proofs. High learning curve.	Partly: not designed for AI, but a codegen AI could use it if extremely advanced. Likely too complex for current models without assistance.

As seen, gp-λ occupies a niche where it intentionally sacrifices some automation (requiring the model to participate in proofs and cost reasoning) in exchange for a simpler, more predictable language structure. We argue this is a good trade-off for the coming generation of AI coding assistants, because it shifts the workload to where AI excels (manipulating

formal symbols, following rules) and away from where AI currently falters (understanding implicit human intentions or dealing with highly irregular patterns).

6 Discussion

Our development of gp-λ opens up several discussion points regarding AI-centric software development and the broader implications of adopting model-focused languages. We highlight a few key themes:

Alignment with LLM Code Generation Pipelines. Gp-λ is designed to slot naturally into an AI-driven coding pipeline. In such a pipeline, an LLM might take a high-level specification (in natural language or tests) and produce gp-λ code. The code is then checked by a verifier (which could be a combination of our type checker, a cost analyzer, and a proof checker). Feedback from the verifier can be fed back to the LLM to refine the code. This loop continues until the code passes all checks, at which point it can be compiled to an executable or linked into a larger system.

This workflow is akin to how a human would iteratively debug and verify their code, but here the process is formal and automated. The advantage of gp-λ in this setting is its *transparency*: every potential issue is exposed as a failed typing or proof obligation, not as a subtle runtime bug or flaky test. For example, if the model forgets an edge case, instead of a test failing (which might not even be written), a proof obligation could fail or a postcondition proof will not go through. The model then has specific guidance on what to fix (it can examine the failing proof goal).

This approach has been partially validated by systems like MoonBit's AI coding assistant, which performs real-time static analysis during code generation:contentReference[oaicite:45]index=45. They report improved code accuracy by guiding the model with parser and type-checker feedback at each step:contentReference[oaicite:46]index=46. Gp-λ enables an even richer feedback mechanism: not just syntax and simple types, but logical correctness and resource usage can be checked on the fly. We envision IDEs where as the AI types each token, a behind-the-scenes gp-λ interpreter is verifying the work. The model, if integrated tightly, can use this to avoid hallucinating impossible code (the "illusion" problem:contentReference[oaicite:47]index=47). In effect, the language's rules become part of the model's stream of consciousness.

Implications for Fine-Tuning and Model Capabilities. One might wonder: do current LLMs have the capacity to produce correct gp-λ code, especially with proofs? The answer is likely "not without further training or specialized prompting." However, gp-λ could serve as an excellent target for fine-tuning because it forces the model to handle formal reasoning. By training on gp-λ code (perhaps generated or extracted from proof assistants), an LLM could improve its ability to do chain-of-thought reasoning for code. Essentially, writing a proof is a chain-of-thought exercise. We hypothesize that an LLM fine-tuned to output both code and proofs will develop better internal consistency and logical reasoning, as it must justify every step.

There is precedent in smaller domains: models fine-tuned for solving math proofs or doing formal logic have shown improved coherence. Gp-λ provides a unified playground where code and math meet. A model trained on it wouldn't treat code as just text to pattern-match, but as something that needs logical closure (e.g., "if I claim X, I must show X"). This could reduce the incidence of confident but wrong code generation that plagues current systems:contentReference[oaicite:48]index=48.

That said, to make this viable, we may need to develop automated tools to generate training data (since currently there is not much gp-λ code out in the wild). We could translate some existing verified programs from Coq/F*/Liquid Haskell into gp-λ. Or we might use the model itself in a bootstrap loop to generate candidate programs and then filter them by proof checking. This is an exciting area of future work: using formal verification as a reward signal or filter for training code models (a form of *reinforcement learning with a proof-checker as the oracle*).

Compiler and Runtime Considerations. From a compiler perspective, gp-λ is relatively straightforward to implement. It could compile to an intermediate language like WebAssembly or to C, similar to how CompCert compiles Clight to assembly with proofs:contentReference[oaicite:49]index=49. The presence of effect annotations can help the compiler optimize or make decisions—for example, pure functions can be memoized or run in parallel safely, IO functions cannot. The cost annotations don't directly change code generation, but they could inform the compiler about whether a loop is bounded by a small constant or by user input, which might influence unrolling or other optimizations.

One novel possibility is using the cost annotations to guide *just-in-time resource allocation*. For instance, if a cost block promises to only use $O(n)$ memory, the runtime can allocate exactly that much space upfront, knowing

it won't overflow. This is particularly useful in embedded or blockchain environments (smart contracts) where you want to pre-charge or limit resource usage to what's been verified:contentReference[oaicite:50]index=50.

Another aspect is the runtime checking of effects: gp-λ could enforce that a function labeled Pure truly doesn't perform IO by sandboxing or by runtime flags (e.g., disallow system calls in pure code). Since we already statically ensure it, this is mostly redundant, but could be a safety net if foreign function calls are allowed.

One challenge is interoperability: how does gp-λ code call into libraries written in other languages? We might need wrappers that have effect types for external calls (e.g., a C function that does IO would be given an IO effect in gp-λ's type). Proof obligations might arise to trust or verify external code's behavior, but that is beyond our current scope. In a fully AI-generated project, perhaps most code is in gp-λ anyway, or any critical logic is.

Human Readability vs Model Readability. By prioritizing model-centric design, we inevitably make code less immediately readable to humans. One could argue this is fine if humans aren't writing it, but what about maintenance and auditing? In safety-critical fields, human audit is important. We believe gp-λ programs, while dense, are still understandable with some effort—arguably more so if you know the formal semantics, because nothing is left mysterious. A proof might be tedious to read, but it's checkable. In contrast, reading a complex C++ program for intent can be harder than reading a gp-λ program plus its proof, since the latter explicitly states all invariants and preconditions.

That said, one could develop tooling to extract a more natural language explanation from a gp-λ program. The proofs themselves could be summarized (since they often follow a certain pattern: e.g., induction on n, etc.). Thus, ironically, having the formal proof might allow automated generation of documentation explaining the code's correctness. This might make human auditing easier than for unverified code.

However, the main target remains AI systems: if the code is reliable, humans might rarely need to dive in except to double-check the model's work in critical cases. Over time, trust might shift from code-as-text to code-as-certificate (the proofs and types being that certificate). This is speculative but aligns with the long-term vision of correct-by-construction software.

Future Verification Pipelines and AI. Gp-λ suggests a future where the gap between programming and verification is closed by AI. Humans currently often write a program then separately try to verify it (if at all). It's labor-intensive, hence not widespread. But an AI has no issue doing repetitive labor, so it can intermix coding and proving. Our pipeline could be extended: imagine an AI writes a gp-λ program, proves key properties, compiles it, and even monitors it at runtime, all autonomously. The role of the human might then be to specify what they want in higher-level terms and to interpret the proofs and results.

In such a scenario, languages like gp-λ serve as the *lingua franca* between the model and the verification tools (and indirectly, the requirements). We foresee integration with property-based testing as well: the model could generate gp-λ code and QuickCheck properties (to test non-critical properties that are easier to test than prove). The code passes tests and proofs, providing a multi-faceted assurance.

Another angle is using gp-λ for *verified learning*: If an AI algorithm (like a new machine learning technique) is implemented in gp-λ, one could prove convergence or complexity claims about it, bridging ML and PL. An AI might even modify its own code in gp-λ and re-verify, leading to adaptive, yet verified, systems.

Finally, we should mention the potential of combining gp-λ with *Automated Amortized Resource Analysis (AARA)* techniques beyond what we did. A recent extension of AARA handles probabilistic programs, for instance; an AI writing a probabilistic algorithm could use an extended gp-λ to also ensure bounds on expected runtime or memory. The possibilities are broad.

7 Conclusion

We presented gp-λ, a programming language oriented towards generative AI systems as the primary "developers." Gp-λ shifts many responsibilities traditionally shouldered by human programmers (like ensuring correctness and efficiency) into the realm of formal language constraints that an AI must satisfy. This model-centric approach is timely: with the growing scale of AI-written code, we need languages that naturally accommodate verification and prevent common pitfalls of AI generation (such as hallucinations and subtle bugs).

Our contributions in this paper are threefold: (1) We motivated the design of gp-λ by analyzing the limitations of human-centric languages in an

AI coding context and grounding our solution in recent research observations (security issues in LLM-generated code:contentReference[oaicite:51]index=51, the need for better AI-tool communication:contentReference[oaicite:52]index=52, etc.). (2) We defined gp-λ formally, including a novel integration of cost annotations and proof obligations into a statically typed, effectful language. We mechanized its semantics and proved soundness properties, ensuring that these features are not only intuitive but enforceable (the cost soundness proof, in particular, provides a rigorous foundation for trusting resource bounds). (3) We demonstrated via examples and comparisons that gp-λ is expressive and practical: it can handle typical programming tasks with additional guarantees, and it aligns well with or improves upon aspects of languages like Liquid Haskell, Rust, and F* when viewed through the lens of AI-generation suitability.

Looking forward, gp-λ serves as a stepping stone toward a future where AI-driven software development is not a wild, unchecked generation of code, but a disciplined process yielding software with built-in correctness guarantees. By teaching AI to "think in gp-λ," we are, in essence, teaching it to reason about its own code. This could significantly reduce the burden of debugging and testing AI-generated software, as many errors will be caught and corrected during generation.

There are several avenues for future work. On the theoretical side, extending gp-λ with richer type system features (e.g., refinement types or simple dependent types) could increase its expressiveness while still keeping it amenable to LLM generation. We deliberately started with a moderate design to see how far we get, but gradually one could incorporate more from the world of proof assistants as models become more capable. Another extension is supporting concurrency and verifying properties like deadlock-freedom or security protocols—GP-'s effect system might be extended to handle such concerns (e.g., an effect for "uses lock L"), combined with proof obligations to show correct usage.

On the practical side, building an actual gp-λ toolchain and an IDE that interfaces with an LLM would allow us to test these ideas in practice. One exciting project would be to modify an existing LLM (like Codex or CodeGPT) via fine-tuning or plugins to produce gp-λ code given problem descriptions. We could then measure, for example, how often the model's first attempt passes the verifier versus how many iterations it needs. That would give insight into the strengths and weaknesses of the approach. Even if the model struggles initially, the existence of a formal verifier in the loop can drive improvements (through reinforcement learning or direct feedback).

Another empirical angle is to reimplement some known algorithms or

systems in gp-λ using AI assistance, to evaluate the readability and performance overhead. For instance, could we have an AI re-generate a portion of the Linux kernel in gp-λ such that it is formally verified for memory safety and some functional properties? That's ambitious, but it's the kind of end goal we have in mind—AI not just generating code, but generating verified, efficient code for complex systems.

In conclusion, gp-λ represents a convergence of trends in AI and programming languages: it treats code as not just something to be written, but something to be *proven*. By requiring the AI to prove its code correct and efficient, we move towards a paradigm of *trustworthy AI programming*. The ultimate vision is a world where software—much of it written by AI—comes with the guarantees we expect from critical systems, yet is developed with unprecedented speed and adaptability. We believe gp-λ is a step in that direction, demonstrating that languages for the AI era can be both model-friendly and mathematically rigorous.

References

[1] Jingxuan He. Moonbit: a programming language for ai-native application (blog post), 2023. URL: https://www.moonbitlang.com/blog/ai-coding, visited on July 7, 2025. :contentReference[oaicite:53]index=53:contentReference[oaicite:54]index=54

[2] Steve Jones. Will LLMs kill Python? (Medium article), Feb. 2023. URL: https://blog.metamirror.io/will-llms-kill-python-c805c8609d11, visited on July 5, 2025. :contentReference[oaicite:55]index=55:contentReference[oaicite:56]index=56

[3] Jan Hoffmann and Steffen Jost. Two decades of automatic amortized resource analysis. *Math. Structures in Comp. Sci.*, 32(8):1–31, 2022. :contentReference[oaicite:57]index=57:contentReference[oaicite:58]index=58

[4] Shih-Chieh Dai, Jun Xu, and Guanhong Tao. A comprehensive study of LLM secure code generation. *arXiv:2503.15554*, Mar. 2025. :contentReference[oaicite:59]index=59

[5] Ralf Jung et al. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, 2018. :contentReference[oaicite:60]index=60

[6] Niki Vazou et al. Refinement types for Haskell. In *Proc. ICFP*, pages 269–282. ACM, 2014. :contentReference[oaicite:61]index=61:contentReference[oaicite:62]index=62

[7] Nikhil Swamy et al. Dependent types and multi-monadic effects in F*. In *Proc. POPL*, pages 256–270. ACM, 2016. :contentReference[oaicite:63]index=63:contentReference[oaicite:64]index=64

[8] UTSA Security Lab. UTSA researchers investigate AI threats in software development (news article), Apr. 2025. URL: https://www.utsa.edu/today/2025/04/story/utsa-researchers-investigate-AI-threats.html, visited on July 4, 2025. :contentReference[oaicite:65]index=65:contentReference[oaicite:66]index=66