Static Type Inference for a Lua-like Language: Foundations and Literature Review

Dominic Bertolo

November 19, 2025

1 Type Inference for a Lua-like Language

1.1 The Volatility of Dynamic Typing

Over the past several decades, dynamically typed programming languages have surged in popularity. Languages such as Python, JavaScript, Ruby, and Lua have been widely adopted due to their concise syntax, rapid prototyping capabilities, and the absence of explicit type declarations. As noted by the creators of Lua, this design philosophy prioritizes "mechanisms" over "policies," allowing for high flexibility during early development [7]. However, while this flexibility accelerates prototyping, it reveals significant costs as projects scale. A defining trait of dynamic languages is that type validation occurs at runtime, meaning errors that could have been resolved statically are only discovered during execution. As established in early static analysis research, inferring types in such environments is notoriously difficult because the flow of values is not explicitly constrained [1]. Consequently, this leads to increased development costs, where bugs manifest only under specific runtime conditions, causing delays and reliability issues.

Lua is a prime example of this volatility. Despite its challenges, it has gained massive traction in the gaming industry due to its ease of embedding, powering logic in titles such as *World of Warcraft*, *Balatro*, and *Angry Birds*. Luau, a high-performance derivative, powers the Roblox platform, serving millions of users [3]. Given this widespread adoption, enhancing Lua's safety through static analysis would yield substantial benefits. However, Lua is particularly difficult to effectively statically-analyze. Its reliance on a single data structure (the table), first-class functions, and metatables presents a "dynamic" barrier that traditional inference engines struggle to process [7]. Lin (2015) categorizes these as "exotic" features, noting that metatables specifically complicate the operational semantics required for formal verification [8].

1.2 The Gap in Existing Solutions

While there have been attempts to introduce static typing to Lua, current solutions prioritize developer convenience over theoretical abidance. Industry standards for dynamic languages, such as TypeScript, often utilize "erasure" semantics, where types are removed at runtime and safety is not guaranteed if data flows in from external sources [2]. In the Lua ecosystem, Typed Lua introduces gradual typing but struggles to model table mutation accurately. Similarly, Luau implements a "non-strict" mode designed to minimize false positives rather than guarantee correctness, explicitly trading soundness for usability [3]. These systems often lead to either overly permissive inferences that fail to catch errors or require extensive manual annotations that negate the concise nature of the language. There remains a lack of a solution that offers a ”pure” inference model—one that guarantees safety without requiring the developer to manually annotate every variable.

The goal of this project is to design and implement ”MiniLua,” a language subset paired with a pure Hindley-Milner type inference system. Unlike existing gradual approaches, MiniLua prioritizes *soundness* and *completeness*. Following the definitions provided by Pierce (2002), the system aims to satisfy the preservation and progress theorems, ensuring that if a program type-checks, it will not get ”stuck” at runtime [9]. By restricting the most volatile features of Lua—specifically global state mutation and dynamic metatable assignment—MiniLua seeks to prove that a Lua-like language can be statically verified with the same rigor as functional languages like Haskell, providing a foundation for robust software development in the Lua ecosystem.

2 Theoretical Framework

2.1 The Hindley-Milner Standard

The project utilizes the Hindley-Milner (HM) type system as its foundational mathematical framework. HM allows for the automatic deduction of the most general type (principal type) of expressions without requiring explicit type annotations [6]. This effectively separates it from the ”gradual” or ”optional” typing systems found in industry standards like Luau, aiming instead for total static verification.

The HM system relies on two critical properties that MiniLua aims to replicate:

1. **Soundness:** Guaranteeing that well-typed programs cannot produce type errors at runtime (often summarized by the slogan ”Well-typed programs cannot go wrong”) [9].
2. **Let-Polymorphism:** Enabling functions to operate generically over different types, provided the polymorphism is introduced via a `let`-binding (or in Lua's case, a `local` declaration) [4].

While standard Lua allows variables to change types dynamically (e.g., `local x = 5; x = "hello"`), this behavior is incompatible with the principal type property of HM. Therefore, MiniLua adopts a strict discipline where variables are immutable in their type assignment, leveraging HM's inference capabilities to maintain flexibility without sacrificing the rigorous safety guarantees defined by Damas and Milner.

2.2 Constraint Generation and Unification

Rather than using the original ”Algorithm W” which interleaves inference and substitution, this project adopts the two-phase architecture proposed by Wand (1987). This approach separates the inference process into two distinct stages: **Constraint Generation** and **Constraint Solving** (Unification) [10].

- **Constraint Generation:** The type checker traverses the Abstract Syntax Tree (AST) of the MiniLua program. For every expression, it generates an equation relating type variables. For example, a function call $f(x)$ generates a constraint $T_f = T_x \to T_{result}$. As noted by Aiken and Murphy, this phase is purely syntactic and independent of the solving logic [1].
- **Unification:** The solver takes the set of constraints and attempts to find a substitution that satisfies all equations. This project will implement a variation of the Robinson unification algorithm to determine

the Most General Unifier (MGU). As detailed in implementation guides by Gräber, this phase is responsible for detecting type mismatches (e.g., trying to unify ‘int‘ with ‘string‘) and reporting them to the user [5].

By adopting Wand’s separation of concerns, the MiniLua type checker ensures that error reporting can be localized. If unification fails, the engine can identify exactly which constraint was violated, providing the developer with precise feedback—a feature often lacking in ad-hoc type systems.

References

[1] Alexander Aiken and Brian R. Murphy. Static type inference in a dynamically typed language. In *POPL ’91: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 279–290, New York, NY, USA, 1991. ACM.

[2] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *ECOOP 2014: Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer.

[3] Lily Brown, Andy Friesen, and Alan Jeffrey. Position paper: Goals of the luau type system. In *HATRA ’21: Human Aspects of Types and Reasoning Assistants*, pages 1–7, New York, NY, USA, 2021. ACM. arXiv:2109.11397v1 [cs.PL].

[4] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL ’82: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212, New York, NY, USA, 1982. ACM.

[5] Martin Gräber. Algorithm w step by step. Technical report, Institut für Informatik, Ludwig-Maximilians-Universität München, 2011. Available at `https://raw.githubusercontent.com/mgrabmueller/AlgorithmW/master/pdf/AlgorithmW.pdf`.

[6] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[7] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Lua: An extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[8] Hanshu Lin. Operational semantics for featherweight lua. Master’s thesis, San Jose State University, 2015.

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

[10] Mitchell Wand. A simple algorithm and proof for type inference. In *POPL ’87: Proceedings of the 14th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 215–222, New York, NY, USA, 1987. ACM.