1 Introduction: Type Inference for a Lua-like Language

1.1 The Rise of Dynamically Typed Languages

Throughout the last few decades, programming languages that have a dynamic type system have gained significant popularity. Languages such as Python, JavaScript, Ruby and Lua have been widely adopted, with their popularity stemming from their focus on concise syntax, rapid prototyping capabilities, and that types are not explicitly declared. However, this flexibility for rapid prototyping and simplicity reveals its cost when project exceeds the prototyping phase.

A dynamic type system refers to when type validation is performed at runtime instead of at compile time. This behavior can lead to runtime errors that could have been discovered and remedied during the development phase if a static type system were in place. Additionally, these class of runtime errors can be difficult to debug as their very nature means that they only manifest during execution, often only under specific conditions. This can lead to increased costs to development teams in the form of time spent debugging and fixing these issues, as well as potential delays in project timelines.

1.2 Static Analysis as a Solution

Static analysis allows for a powerful solution to the problems caused by the nature of dynamic type systems. By analyzing code without requiring execution, static analysis tools can algorithmically verify type constraints and confirm the abidance of code to a set of type rules. This implementation of static analysis can help catch potential type-related errors early in the development process, reducing the likelihood of runtime errors and improving overall code quality, maintainability, and reliability. It forms a hybrid approach that combines the flexibility of dynamic typing with the safety and reliability of static typing.

1.3 The Importance of Lua

The subject of this research is the Lua programming language, a light-weight, high-level, and dynamically typed language. Due to its highly performant nature and ease of embedding, it is widely used in various domains, including video game development, embedded systems, and scripting within larger applications. It's popularity within the video game industry is particularly notable, with titles such as *World of Warcraft*, *Balatro*, and *Angry Birds* utilizing Lua for scripting game logic and behavior. Additionally, a high-performance

variant of Lua, Luau, is the primary scripting language for the Roblox platform, a platform hosting millions of active users and one of the largest development ecosystems in the world. Given Lua's widespread adoption and its significant role in various applications, enhancing its type system through static analysis can have large benefits for both developers and users.

1.4 Lua's Dynamism and Challenges for Type Inference

Lua's dynamic nature presents a difficult challenge for static analysis, especially concerning type inference. Similar to many dynamically typed languages, Lua does not require explicit type annotations, allowing for variables to hold values of any type and change types at runtime. This flexibility is common amongst popular dynamically typed languages, such as Python and JavaScript. However, this very flexibility complicates the process of static analysis, especially static type inference. The absence of explicit type information means that the static analysis tool must deduce types based on the code's structure and behavior. This behavior can be difficult to properly analyze when taking into consideration the various dynamic features of Lua and common programming patterns used by developers. Features such as first-class functions, dynamic table structures, and core language operation redefinition through the use of features such as metatables, and a mutable global state that can be modified at runtime, all contribute to a severe increase in complexity for static analysis tools attempting to infer types. Additionally, common programming patterns used by developers, such as duck typing and dynamic code generation, further contributes to the challenge of accurately inferring types in Lua programs.

1.5 MiniLua: A Lua-like Language to Research

A static analyzer is unable to effectively predict the full range of possible behaviors given a piece of code due to the dynamic features and programming patterns used in Lua. To successfully implement a static type inference system for Lua, a strict set of coding standards and limitations would have to be imposed on developers to reduce the complexity and dynamism of the language and its runtime behavior. However, these limitations would likely debilitate developers' ability to effectively use the language for its intended purpose. By instead exploring a Lua-like language that retains the core syntax and semantics of Lua, while simplifying or restricting some of its more dynamic features, it becomes possible to design a static type inference system that is both effective and practical. This research explores MiniLua, a subset of the Lua programming language that aims to retain the core syntax and semantics of Lua while simplifying or restricting some of

its dynamic features. By focusing on MiniLua, this research aims to explore the design and implementation of a static type inference system that can effectively analyze Lua-like code that maintains a balance between flexibility and static analyzability.

2 Background: Static vs. Dynamic Typing, and Type Inference

This section provides the necessary background on static and dynamic typing, as well as type inference techniques relevant to the context of MiniLua.

2.1 Static vs. Dynamic Typing

The distinction between static and dynamic typing is primarily defined by when the type-checking is performed.

- **Static Typing**: In a statically typed language, type-checking is performed at compile-time. During this process, variables are attributed to a specific type and these types along with their use-cases are verified before the program is executed. When the compiler encounters a type mismatch, or use cases that conflict with a variable's type, such as `int x = 10; x = "hello";` , the compiler will often raise a type error, preventing the program from compiling until the error is resolved. This approach prioritizes type safety and early error detection.
- **Dynamic Typing**: In a dynamically typed language, type-checking is performed at runtime. Variables are bound to values instead of types, and the values themselves carry implicit type information. In many dynamically typed languages, a variable can often times be reassigned to a value that does not match the type of its previous value. For example, in Python, the following code is valid: `x = 10; x = "hello";` . The type of `x` changes from an integer to a string without any compile-time errors. However, if an operation is performed that is incompatible with the current type of the variable, such as `x + 5` when `x` is a string, a runtime error will occur. The approach prioritizes flexibility and ease of use.

2.2 What is Type Inference?

Type inference is the ability of a compiler to automatically determine the type of expression at compile-time without the need for explicit type annotations. Type inference algorithms do not exist in mutual exclusivity with either static or dynamic typing. They act as a feature layered on top of a type system, most commonly found in statically typed languages.

An example of a statically typed language implementing a form of type inference is the Go programming language. Go is a statically typed language that allows for type inference through the use of the `:=` operator. For example, in the following code snippet, the type of `x` is inferred to be an integer based on the value assigned to it:

```
x := 10 // x is inferred to be of type int
```

In this case, the compiler infers the type of `x` to be an integer based on the value assigned to it, without requiring an explicit type declaration or annotation.

Another example of statically typed language implementing type inference is the Haskell programming language. Haskell uses a powerful type inference algorithm called Hindley-Milner, which allows for the automatic deduction of types for expressions based on their usage and context within the code. For example, in the following Haskell code snippet, the type of the function `add` is inferred to be `Int → Int → Int` based on its definition:

```
add x y = x + y // add is inferred to be of type Int → Int → Int
```

In this case, the compiler infers the type of `add` to be a function that takes two integers as arguments and returns an integer, without requiring explicit type declaration or annotation.

An example of a dynamically typed language implementing a form of type inference is TypeScript, a superset of JavaScript that adds static typing capabilities. TypeScript uses type inference to automatically deduce the types of variables and expressions based on their usage and context within the code. For example, in the following TypeScript code snippet, the type of `x` is inferred to be a number based on the value assigned to it:

```
let x = 10; // x is inferred to be of type number
```

In this case, the TypeScript compiler infers the type of `x` to be a number based on the value assigned to it, without requiring an explicit type declaration or annotation.

2.3 Approaches to Typing Dynamic Languages

Where it concerns dynamically typed languages, implementation of a type inference system is often approached through one of two main methodologies:

1. **Algorithm-Based (Pure) Inference:** One of the most well-known algorithm-based typed inference systems is the Hindley-Milner type system, which is used in languages such as ML and Haskell. The goal is that types can be inferred through a set of rules and algorithms that analyze the structure and behavior of the code. In this approach, so long as code adheres to the rules and constraints defined by the type system, types can always be accurately inferred without the need for any additional type annotations or hints from the developer. Due to its derivation from formal logic and type theory, it is often considered to be theoretically sound, complete, and "pure". However, the strictness of rules and constraints necessary for this approach to function often limits its applications to many programming languages where their use cases and features demand more flexibility and less rigidity.

2. **Gradual / Optional Typing:** A modern and pragmatic approach that has achieved significant adoption in recent years is the use of gradual typing, also known as optional typing. Popular systems such as TypeScript, Sorbet, and Luau are gradual type systems. This approach is a compromise compared to pure algorithm-based inference systems. It allows developers to optionally add type annotations to their code, individually selecting which sections of code should be forced to abide by the static type system. Most approaches of gradual typing implement this gradual or optional typing by providing a universal `any` type that can be used to annotate variables or expressions that should not be subject to type inference. This type then is considered generally compatible to any other type, resulting in the variable or expression being exempt from static type checking. The type can then be provided to values implicitly unless otherwise specified by a type annotation, or is provided to the developer as a type to be explicitly cast to a value or expression. The approach allows for a balance between flexibility of dynamic typing and the safety of static typing, enabling developers to incrementally adopt static typing case by case as needed.

2.4 Process of Type Inference: Constraints and Unification

Modern approaches to type inference often involve the use of constraint-based inferences that separate the process of type inference into two main phases: constraint generation and constraint solving (unification).

- **Constraint Generation:** The type system algorithm traverses the abstract syntax tree (AST) of the code. During this traversal, for every snippet of code that can be analyzed for an attributed type, a variable is created to represent the type of that snippet. As the type system continues to traverse the AST, it generates a list of constraints that represent the relationship between the types of different snippets of code. For example, if a function is called with an argument, a constraint is generated that states that the type of the argument must be compatible with the type of the parameter defined in the function's signature.
- **Constraint Solving (Unification):** After the traversal of the AST is complete, and all constraints have been generated, the type system then enters the constraint solving phase, often referred to as unification. During this unification phase, the type system attempts to find a corresponding language type for each type variable generated during the constraint generation phase that allows all the corresponding constraints to evaluate to true. If no mapping can be found that satisfies all constraints, a type error is raised. This error indicates that there exists use cases that result in a conflicting constraint that cannot be satisfied given any possible type assignment.

This process can be demonstrated with the following simple example in a Lua-like pseudocode:

```
function f(x)
 return x + 1
end
result = f(5)
```

The first step is to assign type variables to each expression:

- `f`: T_f
- `x`: T_x
- `+`: T_+

- `1`: T_1
- `5`: T_5
- `x + 1`: T_{exp}
- `result`: T_{result}

The second step is to generate constraints based on the operations and function calls:

- From the literal `1`, we have the constraint: $T_1 = \text{number}$
- From the literal `5`, we have the constraint: $T_5 = \text{number}$
- From the addition operation `x + 1`, we have the constraint: $T_{exp} = T_x + T_1$
- From the function definition `function f(x)`, we have the constraint: $T_f = T_x \to T_{exp}$
- From the function call `f(5)`, we have the constraint: $T_f = T_5 \to T_{exp}$
- From the assignment `result = f(5)`, we have the constraint: $T_{result} = T_{exp}$

The third step is to solve the constraints through unification:

- System
  - $T_1 = \text{number}$
  - $T_5 = \text{number}$
  - $T_{exp} = \text{number}$
  - $T_{result} = \text{number}$
- Solving
  - Substitute T_f from function definition into function call constraint: $T_x \to T_{exp} = T_5 \to T_{exp}$
  - Substitute T_5 with number: $T_x \to T_{exp} = \text{number} \to T_{exp}$

Final Inferred Types: The system has successfully inferred the following types:

- `f`: number → number
- `x`: number
- `result`: number

3 Core Literature Section: Constraint-Based Typing

As previously mentioned, constraint-based typing is a widely adopted approach for implementing type inference systems, especially in dynamically typed languages. This section explores the Hindley-Milner type system, which will serve as the foundational basis for the type inference system designed for MiniLua.

3.1 Hindley-Milner Type System

The Hindley-Milner (HM) type system, also known as Damas-Milner type system, is one of the foundational algorithms for type inference in statically-typed functional programming languages. It was first implemented in a programming language was the ML (Meta Language) programming language, then later adopted by other languages such as Haskell and OCaml.

The HM type system is based on the concept of polymorphism, where values and expressions can be described as having polymorphic types. A type being polymorphic meaning that its type can be generalized to represent multiple specific types. The generalization of types allows for greater flexibility and code reusability, as functions and expressions are allowed to operate on a wider range of types without explicit handling of each specific type. However, monomorphic types, types that represent a single specific type, are also supported in the HM type system. Polymorphic types can be deduced to monomorphic types through the use of constraint generation and unification to require specific types in certain contexts. The resulting type after unification is referred to as the principal type, which is the most general type that satisfies all constraints for a given variable type.

At the time of its first implementation, in ML, the HM type system was contradictory to the popular type systems of the time in languages such as Pascal and C. These type systems only supported monomorphic types, requiring explicit type annotations for all variables and expressions. This came at a benefit of human readability for how memory was being laid out for various types and variables, a strong quality for systems-level programming. However, it limited the flexibility of the language as functions and expressions could only operate on specific types per implementation. This meant that for many of the these monomorphic language, true abstraction and generalization of code was difficult if not impossible to achieve.

Successors to some of these languages, such as C++ and Java, later implemented forms of polymorphism through the method of subtyping. This method of subtyping allowed for a concept called overloading, where multiple implementations of a function or method could be defined for different specific types. However,

this approach still required explicit type annotations and handling for each specific type, leading to code duplication and increased complexity. Additionally, this approach did not allow for true polymorphism, as functions and expressions could not operate on a wider range of types without explicit handling of each specific type. An implementation of the HM type system's approach to polymorphism would later result in the development of generics in many languages such as C++ and Java, allowing for true polymorphism and code generalization.

3.2 Let-Polymorphism

The decision to generalize types for variables and expressions must have guidelines to ensure that the type system remains sound and consistent. In the case of parameters of a function, the type must ensure that it is not polymorphic in a way that would allow for conflicting types to be used in different contexts. The way in which the HM type system handles this is through the use of let-polymorphism.

Let-polymorphism is the decision to only allow for polymorphic types to be generalized at let-bindings. A let-binding is a construct that allows for the definition of a variable and its associated value or expression. By restricting the generalization of types to only occur at let-bindings, the HM type system ensures that polymorphic types are only introduced in a controlled manner, preventing conflicting types from being used in different contexts. This restriction helps maintain the soundness and consistency of the type system, as it prevents the introduction of polymorphic types that could lead to type errors or inconsistencies. Type variables that are not bound by a let-binding are considered free type variables, and are treated as monomorphic types during the type inference process. This means that they cannot be generalized to polymorphic types, and must be assigned a specific type based on their usage and context within the code.

3.3 Hindley-Milner for Functional Languages

Functional programming is a programming paradigm that treats computation as the representation of mathematical functions. It is a pure form of programming that emphasizes immutability and deterministic behavior. It achieves this deterministic behavior through the use of data being immutable, meaning that once a value is assigned to a variable, it cannot be changed. This immutability ensures that functions always produce the same output for the same input, leading to predictable and reliable behavior. Functional programming languages often support higher-order functions, which are functions that can take other functions

as arguments or return functions as results. This allows for greater abstraction and code reuse, as functions can be composed and manipulated in a more flexible manner.

It is through these characteristics of functional programming that the HM type system was able to be effectively implemented in functional programming languages such as Haskell. The immutability of data in functional programming ensures that variables and expressions have a consistent type throughout their lifetime, making it easier to infer types based on their usage and context. Additionally, the use of higher-order functions allows for greater abstraction and code reuse, which aligns well with the polymorphic nature of the HM type system. The deterministic behavior of functional programming also ensures that functions and expressions can be analyzed in isolation, without the need to consider the broader context of the program. This allows for more accurate type inference, as the type system can focus on the specific behavior of individual functions and expressions.

3.4 Hindley-Milner for Imperative/Dynamic Languages

Contrary to the functional programming paradigm, imperative programming is a programming paradigm that focuses on describing how a program operates through a sequence of statements that change a program's state. Unlike functional programming, imperative programming allows, even encourages, the use of mutable states. This mutability allows for variables to be reassigned to different values throughout their lifetime, leading to more complex and dynamic behavior. Imperative programming relies on control flow constructs, such as loops and conditional statements, to determine the order of execution and the flow of data within a program. While allowing for greater flexibility and dynamic behavior, it is this strength that complicates the implementation of a type inference system based on the HM type system.

The mutability of data in imperative programming means that variables and expressions can change their type throughout their lifetime, making it more difficult to infer types based on their usage and context. Additionally, the use of control flow constructs introduces additional complexity, as the type system must consider the broader context of the program when analyzing functions and expressions. This can lead to situations where the type of given variables or expressions is dependent on the specific path taken through the program, making it more challenging to accurately infer types.

3.5 Hindley-Milner for Lua-like Languages

Lua is described as a multi-paradigm programming language, supporting imperative, functional, procedural, and object-like oriented programming styles. Despite this multi-paradigm nature, Lua's core design and usage patterns are primarily imperative. This imperative nature of Lua introduces challenges for implementing a type inference system based on the HM type system, as previously discussed. The extent of Lua's dynamic features, such as a dynamic table structures and a mutable global state, result in additional complexity for type inference. Pure type inference systems based on the HM type system often struggle to effectively analyze Lua code due to these dynamic features and programming patterns. To effectively implement a type inference system for a Lua-like language, it is necessary to simplify or restrict some of these dynamic features, as is done in MiniLua. By focusing on a subset of Lua that retains its core syntax and semantics while reducing its complexity, it becomes possible to design a type inference system that can effectively analyze Lua-like code.

4 Related Work: Typed Lua, Luau, TypeScript, and Teal

The HM type system functions on the basis of pure type inference through constraint generation and unification. However, modern programming languages have adopted more pragmatic approaches to type inference, especially when it concerns dynamically typed languages. This section explores some of the most notable implementations of type inference systems in dynamically typed languages, including Typed Lua, Luau, TypeScript, and Teal.

4.1 Typed Lua

Typed Lua aims to bring optional static types to the Lua programming language while preserving its dynamic nature. Its approach to implementing a type inference system for Lua is through the use of purely optional typing, as opposed to gradual typing. In Typed Lua, the default is to have no runtime type checking, solely relying on static analysis to catch errors during development. This approach allows developers to incrementally add type annotations to their code, enabling them to choose which parts of their code should be subject to static type checking. Typed Lua's type system is designed to be sound and consistent, ensuring that type errors are caught during development without introducing runtime overhead. However, this approach can lead to situations where type errors are not caught until runtime if developers choose not to add type

annotations to certain parts of their code. Additionally, Typed Lua's implementation demonstrates that annotations are necessary to achieve soundness in the type system, as pure type inference is unable to maintain soundness and completeness while maintaining Lua's full feature set.

4.2 Luau

Luau is a high-performance variant of Lua developed by Roblox, primarily used for scripting within the Roblox platform. Luau implements a gradual type system that allows developers to optionally add type annotations to their code. This approach enables developers to choose which parts of their code should be subject to static type checking, providing a balance between flexibility and safety. Luau's type system is designed to be sound and consistent, ensuring that type errors are caught during development without introducing runtime overhead. Additionally, Luau's type system includes features such as union types, intersection types, and type aliases, allowing for greater expressiveness and flexibility in defining types. Luau's implementation demonstrates the effectiveness of gradual typing in dynamically typed languages, providing developers with the ability to incrementally adopt static typing as needed. The team behind Luau has described it to be a "type inference engine first and a type checker second", emphasizing the importance of type inference in their design. A unique aspect of Luau's type inference system, compared to a system such as Hindley-Milner, is its use of localizing type inference to specific scopes, allowing for more precise type inference in the presence of mutable state and dynamic features. In Luau's type system, a variable's type can be refined based on control flow analysis, allowing for more accurate type inference in different contexts within the same scope. One of the key limitations of Luau's implementation of a type system is its complete reliance on type annotations for all data structures and functions. A notorious example of this limitation is Luau's inability to infer the type of `self` within methods of tables (objects) without an explicit type annotation. However, these limitations are the result of a pragmatic design choice to prioritize performance and usability over theoretical soundness and completeness that might be seen in a pure type inference system such as Hindley-Milner.

4.3 Teal

Teal is a statically typed dialect of Lua that is trans-piled into standard Lua code. In its implementation, it is entirely annotation-driven. Teal does not serve to provide Lua with a type inference system, but rather to provide a separate development experience for developers who wish to use a statically typed language on top

of Lua. Do to this separation from Lua, Teal is able to implement a consistent static type system without the need to accommodate Lua's dynamic features. However, the drawback of this approach is that the dynamic nature of these features are not represented identically. Teal separates the use of tables to be either records (static structures with defined fields), maps (dynamic structures with key-value pairs), or arrays (ordered collections). This separation allows for a more consistent type system, but deviates from Lua's core design where tables are used as a single, flexible data structure that can serve all these purposes. Additionally, Teal requires explicit type annotations for all variables and expressions, which can lead to increased verbosity and reduced flexibility compared to Lua's dynamic typing. While Teal provides an effective static type system for Lua-like code, it does so but not only using an impure type inference system, but a combination of an annotation-driven type system that is entirely separate from native Lua.

5 Gaps in Existing Literature / Motivation for the Project

5.1 Common Limitations in Existing Work

Amongst the reviewed existing work, several common limitations can be identified that highlight the compromises made in the design of type inference systems for Lua-like languages.

- **Reliance on Annotations:** Many existing systems, such as Typed Lua and Teal, rely heavily on explicit type annotations to achieve soundness and consistency in their type systems. This reliance on annotations can lead to increased verbosity and reduced flexibility, as developers must explicitly define types for variables and expressions. This requirement can be particularly burdensome in large codebases or when working with complex data structures.
- **Limited Type Inference Capabilities:** Some systems, such as Luau, have limitations in their type inference capabilities, particularly when it comes to inferring types in the presence of mutable state and dynamic features. For example, Luau is unable to infer the type of `self` within methods of tables (objects) without an explicit type annotation. These limitations can lead to situations where developers must resort to annotations or workarounds to achieve the desired type safety.
- **Trade-offs Between Soundness and Usability:** Many existing systems make trade-offs between soundness and usability, prioritizing performance and developer experience over theoretical soundness and completeness. While these trade-offs can lead to more practical and usable type systems, they can

also result in situations where type errors are not caught until runtime or where the type system is unable to accurately infer types in certain contexts.

5.2 Cause for Compromises

The compromises observed in existing type inference systems for Lua-like languages can be attributed to several factors:

- **Dynamic Features of Lua:** The dynamic features of Lua, such as dynamic table structures and a mutable global state, introduce significant complexity for type inference. These features make it difficult to accurately infer types based on usage and context, leading to limitations in type inference capabilities.
- **Imperative Programming Paradigm:** Lua's imperative programming paradigm, which emphasizes mutable state and control flow constructs, further complicates type inference. The mutability of data means that variables and expressions can change their type throughout their lifetime, making it more challenging to infer types accurately.
- **Performance Considerations:** Many existing systems prioritize performance and developer experience over theoretical soundness and completeness. This focus on practicality can lead to trade-offs that limit the capabilities of the type inference system.

5.3 Compromises to be Made

Throughout every reviewed implementation of a type inference system for Lua-like languages, none have been able to accomplish both soundness and completeness while fully accommodating Lua's dynamic features and imperative programming paradigm. Completely absent in the reviewed implementations is the presence of a pure type inference system, such as Hindley-Milner. This absence highlights the inherent challenges and trade-offs involved in designing type inference systems for Lua-like languages. To effectively implement a type inference system for MiniLua, it is necessary to make certain compromises that balance soundness, completeness, and practicality.

Teal demonstrates that statically typed Lua-like code can be effectively analyzed through the use of an annotation-driven type system, but at the cost of flexibility and fidelity to Lua's core design. Additionally, it

demonstrates that the compromise of limiting dynamic features does not necessarily debilitate the language's usability for its intended purpose. Lua claims to be a multi-paradigm language, but its core design and usage patterns are primarily imperative. By reducing the use of dynamic features and mutable states, a subset of Lua can be defined to act as a functional programming language that exists for solutions that can be depicted deterministically. This reduction in complexity maintains for a wide range of applications to be effectively developed, alongside maintaining one of the paradigms Lua is advertised to support. This new subset, MiniLua, with its functional programming focus, will allow for the implementation of a type inference system based on the Hindley-Milner type system. By focusing on a subset of Lua that retains its core syntax and semantics while reducing its complexity, it becomes possible to design a type inference system that can effectively analyze Lua-like code while achieving soundness and completeness.

6 Conclusion: How This Review Informs MiniLua's Design

The review of existing literature on type inference systems for Lua-like languages has provided valuable insights that will inform the design of MiniLua's type inference system. The analysis of various approaches, including Typed Lua, Luau, and Teal, has highlighted the challenges and trade-offs involved in designing type inference systems for dynamically typed languages.

6.1 Lessons to Apply to MiniLua's Design

MiniLua's type inference system should be constraint-based and solved using unification. By using the concepts of the HM type system, and borrowing its implementation in languages such as ML and Haskell, MiniLua can achieve soundness and completeness in its type inference system. The implementation of polymorphism should aim to follow the let-polymorphism approach, ensuring that polymorphic types are only introduced in a controller manner at let-bindings. This restriction aligns will with a subset of Lua as the declaration of variables through `local` statements can be treated as let-bindings. Continuing with concepts from the HM type system, MiniLua will borrow the principle of value restriction from implementations such as Standard ML and OCaml. This principle ensures that only expressions that are syntactic values can be generalized to polymorphic types, preventing unsoundness in the presence of mutable state. By enforcing this restriction, MiniLua can maintain soundness in its type system while still allowing for polymorphism in a controlled manner.

6.2 Pragmatic Decisions for Language Grammar

To effectively implement a functional programming driven subset of Lua, certain pragmatic decisions must be made regarding the language grammar. MiniLua should aim to limit or restrict features that are heavily dynamic in nature, or that encourage imperative programming patterns. This includes limiting the use of dynamic table structures and global variables. By reducing the complexity introduced by these features, MiniLua can maintain a more predictable and analyzable codebase, allowing for more accurate type inference. Additionally, MiniLua should borrow from Teal's approach of separating tables into distinct data structures that cannot be interchanged. This separation allows for a more consistent type system, as each data structure can have its own defined type and behavior. By implementing these pragmatic decisions in the language grammar, MiniLua can effectively balance the need for flexibility and expressiveness with the requirements of a sound and complete type inference system.

6.3 Pragmatic Decisions for Type System

A common implementation choice is the use of gradual typing, as seen in Luau. The use of gradual typing contradicts the goal of implementing a pure type inference system based on Hindley-Milner. While it stands as a pragmatic choice to prioritize usability and performance, it introduces compromises that stray from the theoretical soundness and completeness of the type system. MiniLua's type system should avoid the use of gradual typing, instead opting for a pure type inference system based on the Hindley-Milner type system. This decision aligns with the goal of achieving soundness and completeness in the type inference system, while still allowing for flexibility and expressiveness through the use of polymorphism. By avoiding gradual typing, MiniLua can maintain a more consistent and predictable type system, ensuring that type errors are caught during development without introducing runtime overhead. A gradual type system may be accepting of certain unsound behaviors for the sake of usability, but MiniLua's type system should prioritize theoretical soundness and completeness to provide a robust and reliable development experience.

MiniLua's type inference system should prioritize soundness and completeness over abidance to the reference, Hindley-Milner type system. While polymorphism is a core feature of the HM type system, MiniLua should only implement polymorphism where it can be achieved without compromising soundness. This should be demonstrated through the successful implementation of complete and sound monomorphic type inference before extending to polymorphic types. By prioritizing soundness and completeness, MiniLua can provide a

reliable and predictable development experience, ensuring that type errors are caught during development without introducing runtime overhead.

7 Final Synthesis

This literature review of existing work on type inference systems for Lua-like languages has validated the "MiniLua" project in its goal to implement a pure type inference system based on the Hindley-Milner type system. The review has highlighted the challenges and trade-offs involved in designing type inference systems for dynamically typed languages, particularly in the context of Lua's dynamic features and imperative programming paradigm. By analyzing various approaches, including Typed Lua, Luau, and Teal, the review has identified common limitations and compromises made in existing systems, such as reliance on annotations, limited type inference capabilities, and trade-offs between soundness and usability.