TypeLoom: Gradual Typing with the LSP

A Preprint

Adhyan H.
Department of Engineering
University of Saskatchewan
Saskatoon, SK S7N 5A9
jcf879@usask.ca

June 11, 2024

Abstract

This paper introduces TypeLoom, a tool utilising a novel approach to add gradual, optional typing into legacy code bases of dynamically typed languages. TypeLoom leverages the Language Server Protocol (LSP) to provide in-editor type information through inlay hints and collect subsequent type information through code actions to type information. This approach differs from the ones that exist in so far as it requires no syntactical changes to add type hints (like in Python, TypeScript) and it does not require syntactically correct comments to provide type information (like in @ts-docs and Ruby). TypeLoom utilises a graph based data structure to provide type inference and type checking. This graph-based approach is particularly effective for gradual typing as it allows flexible representation of type relationships and dependencies

Keywords Pluggable Types · Gradual Types · Optional Types · LSP · Graph

1 Introduction

In computer science, a type system serves as a framework designed to attribute a type to various program components, such as variables, expressions, and functions. The primary goal of a type system is to reduce errors in computer programs by ensuring consistent interfaces and verifying their connections. There are two broad categories of type systems: static and dynamic. Static type systems perform type checking at compile time, dynamically typed languages defer this process until runtime. Proponents of static typing argue that the stronger type checking helps in catching a lot of trivial bugs even before the program is executed, while proponents of dynamic typing argue in favour of the flexibility and rapid iteration the system allows for. These two paradigms are not mutually exclusive, and many programming languages incorporate elements of both static and dynamic typing. The dichotomy between static and dynamic typing continues to shape modern programming discourse, offering developers a spectrum of tools to balance safety, performance, and ease of use based on their specific needs.

Around the 1990s, we observe the emergence of dynamic typing. Languages like Lua, Bash, Python, JavaScript, PHP etc. came around. Dynamic typing became synonymous with scripting languages, allowing developers to discard type annotations and focus on writing programs and shipping code instead of fussing over types. As type systems have enhanced and become more powerful, the programming landscape has shifted once again in favour of static typing. Type systems like the Hindley-Milner type systems make it so that type annotations do not always need to be explicitly stated. Linear Types, Generic Associated Types, and Higher Order Types allow such static system to be more flexible and expressive. Another issue that static types have solved is that of scale. Static type information can serve as a form of self-documentation and help increased maintainability and allow developers to refactor with increased confidence. A new hybrid paradigm that aims to combine the best of the both worlds is gradual typing. Gradual typing takes a pragmatic approach to type systems by allowing developers to add type information for certain variables and expressions which is checked at compile time while allowing some parts of the code base to remain un-typed.

2 How do we type gradually?

In this section we will briefly explore the different implementations of gradual type systems. This is by no means an exhaustive list but should be enough to provide a basic understanding of the concepts we will build upon.

2.1 Transpilers

Transpiler or a source to source compiler simply put is a compiler that takes as its input a programming language and outputs or compiles to another programming language. If we have a language that is dynamically typed and does not support type annotations, one approach to adding type annotations would be to create a super set of the language that supports said type annotations and statically checking them at compile time, the transpiler then compiles down to a dynamic language while type checking to ensure type errors are not encountered during program execution. This is the approach the Microsoft took when it introduced TypeScript in 2012. This approach is especially beneficial when the dynamic language is widely used and for any super set to succeed interoperability is a must for adoption.

2.2 Optional Syntax

Another popular approach is introducing optional syntax into the language. Python in 2015 introduced type hints which is additional syntax that allows programmers to add type annotations and then leverage external type checking tools like mypy or pyre. The interpreter or the compiler itself does no type checking. It is important to acknowledge that this approach does break backwards compatibility and may requires changes to the compiler to support the new syntax but comes with the advantage of not introducing another language into the development cycle.

2.3 Standardised Comments

An approach similar to the optional syntax approach is to introduce type annotation syntax but making use of comments. This immediately has the benefit of making changes to the language's syntax or making changes to the compiler to accommodate the aforementioned syntax changes. The type annotations in comments are similarly consumed by external type checkers. This approach can be seen with @ts-docs and Ruby's initial foray into static types. A major drawback of this approach is that developers expect comments to be ignored by the programming language and it may be awkward for a lot of them to see type errors in comments.

2.4 So, now what?

Maybe we can establish another approach to gradual typing which avoids the following pitfalls that we have seen in the previous approaches, mainly:

- Avoid the need for a super set or maintaining a transpiler.
- Avoid the need for new syntax or changes to the existing compiler.
- Avoid making the comments semantic or making them have a definite syntax.

3 Exploring a potential solution

I shall be implementing a type system for the Suneido programming language mainly because I have worked with it previously in a professional capacity and have contributed to its open source compiler of which I have a decent understanding. Suneido is an integrated application platform that combines an object-oriented programming language, a client- server relational database, and various application frameworks. It includes an IDE and supports networked applications, all as an Open Source project available for free. Suneido aims to be a simple, open, and lightweight alternative to complex, expensive technologies like Visual Basic and SQL Server.

3.1 Gathering Type Information

For a lot of languages even those which are dynamically typed, some amount of information can be easy to extract. For example, in the following line of code. It is quite easy to imagine that the type of x must be of some number type.

```
x = 123
```

We aren't quite sure if it is a u8, u32, i32, i8 or some other more narrower type. But as it is the case with gradual type systems such narrowing and use of such concrete types is not always possible. Thus, when we are crafting a gradual type system it is important to choose our types carefully. For now let's just consider the type *Number* and we can say that our type system now knows that the variable *x* is of type *Number* and cannot be assigned a value of another type which is not compatible with the the type *Number* in other words, *x* can only be assigned a value which has a sub-type or alias of the type *Number*.

```
x = 123       # Initial type inference
x = -49.690 # this is okay
x = "hello" # this is NOT okay
```

3.2 Displaying Type Information

Now with the limited type information we have, we can start displaying it to the user via the Language Server Protocol (LSP). Inlay hints are special markers that appear in the editor, they are typically rendered as virtual text and cannot be edited by the developer. Most modern IDEs like: Neovim, VSCode, and IntelliJ IDEA support Inlay Hints. For our tool the inlay hints look something like the following.

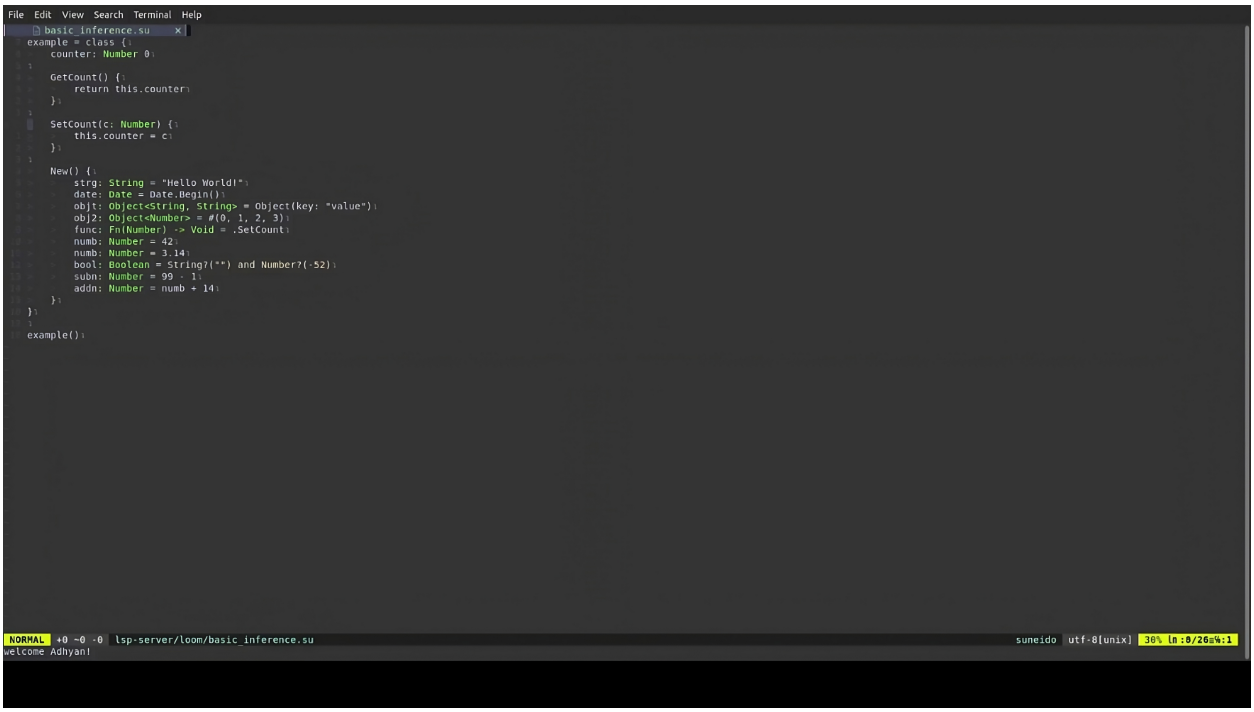

Figure 1: An example of TypeLoom displaying basic types using Inlay Hints

The types we shall use for our systems resemble the primitive types that Suneido offers mainly:

- Number
- String
- Date
- Objects
- Functions
- Boolean

In my opinion, Objects are the most interesting and powerful types in Suneido. Objects are general purpose containers and can be used as: arrays, lists, records, hashmaps. In fact, Objects are also used to represent instances of classes internally. Our gradual type system extends the list of types that already exist by adding the following:

- Any
- Unknown
- Never
- Union
- Intersection
- Void

These are very useful utitlity types, *Any* can be used where some variables need to still by dynamically typed, *Unknown* can be used for a variable whose type is not yet known. *Void* can be used to indicate a function does not return anything and so on.

3.3 Checking Type Information

Implementing a type checker is by no means an easy feat, and as I started to look into type systems the analogy that made the most sense to my head when reading about bi-directional type checking is a graph. There has already been some research into this field of representing types in a graph A graph where each node represents a variable and is tagged with its type information. An edge between two nodes represent a type constraint in this case a type equality. If two nodes *x* and *y* are connected, we can be confident that they are equal or at least compatible in their types. This very simple type checking paradigm now empowers us to do some basic type inference. Say we have two nodes *x* and *y* which are connected and the type of *x* is known to be of type *Number* then we can to a degree of certainty say that *y* must also have a type that is at the very least compatible with the type *Number*. The TypeLoom type system comprises of two steps: Inference and Checking. The first step, type inference, where the system tries to narrow types if possible, for example, if a node of type *Unknown* is connected via an edge to a node that has a more concrete type let's say type *String* then we can safely change the type of the *Unknown* node to *String*. Now we are able to do some simple type checking. Type checking is performed simply by checking if a path exists between two base types, if it does a type errors has occurred as a variable cannot have two base types at the same type. The term base types refers to all the primitives our type system supports and some custom user defined types. For example, a variable *foo* cannot be *Number* and *String* at the same time. A variable *foo* however, can have the types *Number* and *i32* at the same time because they are compatible types.

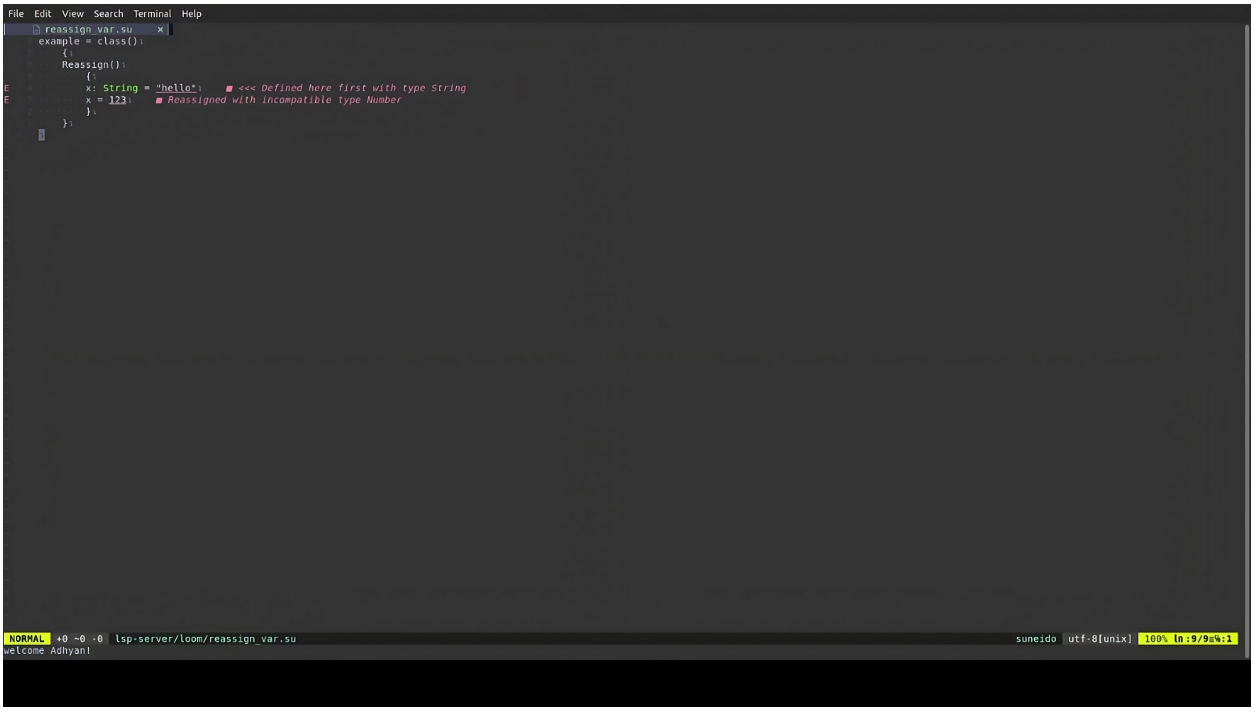

Figure 2: An example of TypeLoom detecting a type error where variable is reassigned

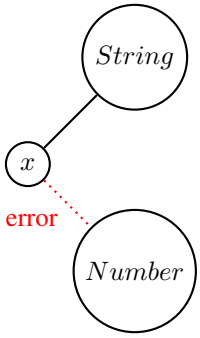

Figure 3: A representation of graph that detects a simple type mismatch

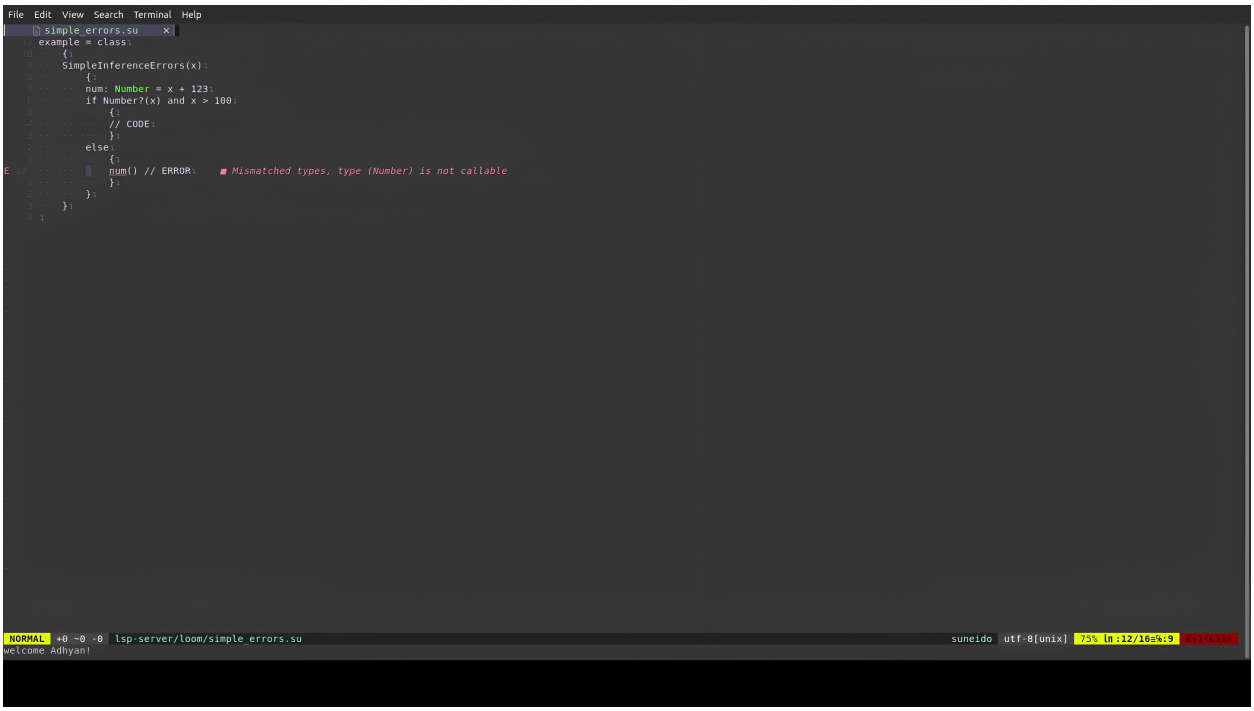

Figure 4: An example of TypeLoom detecting a type error where a type Number is being called as a Function

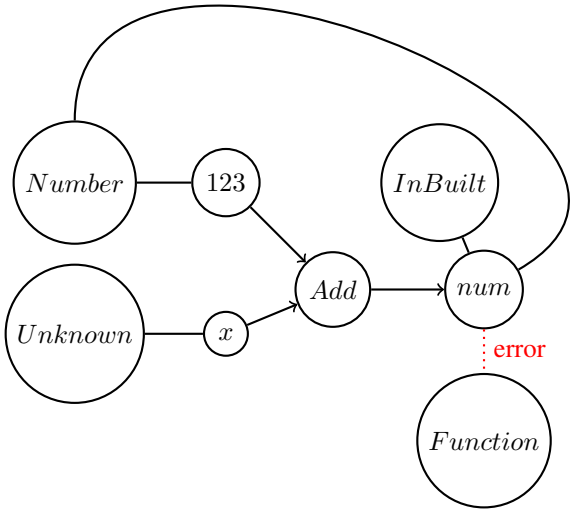

Figure 5: A representation of graph that detects a more complicated type mismatch

In the figure 4 and figure 5 the graph., we can observe a more involved example of catching type errors. This is an excellent example of both inference and checking at work. The function *SimpleInferenceErrors* takes an an input *x*, we are unaware as to what the type of that argument is so we label it as of type Unknown by connecting the node that represents the variable *x* with the node that represents the type *Unknown* itself. On the second line of code we encounter

```
num = x + 123
```

From the above we are able to infer that *x* must be of type *Number*, we know that *123* is of type *Number*. The + operator in Suneido only acts on numbers. And we know that the + operator returns the type *Number* and so *num* is of type *Number*. Since *num* is a variable, it is first marked with the type *Unknown* which is then narrowed to *Number*. As seen in the graph, the nodes are constructed for the aforementioned and connected. Please note that the directional arrows represents input and output of a function, *Add* has two inputs and one output. Later, in the code we see that *num* is being called as a function using the *()* syntax. Our type system connects the node for *num* to the primitive type node for the type *Function*, when we run our type checking algorithm which is a simple depth first search between all the possible primitive types. The type checking algorithm sees that a path between the nodes *Function* and the node *Number* exists, since we know these are incompatible with each other we raise an error.

3.4 How do we construct type graphs?

To construct a type graph, the TypeLoom system takes in the source code and using the Suneido compiler, compiles them down to an s-expression format. After doing so, the loom-compiler generates byte code which upon execution produces the graph. Byte-code is used instead of an AST (Abstract Syntax Tree) for the same reason the sqlite engine uses byte code instead of an AST. The byte code that is seen in 6 is simplified, as the actual byte-code output uses UUID (Universally Unique Identifiers) and name mangling as it is possible to have similarly named variables across two methods in a class or in multiple methods across different classes. Suneido has only two scopes: global and local. This also makes the work of the type checker a bit simpler as all variables within a method are confined to it and braces do not define a new scope.

Figure 6: An example of source code, s-expression format, and byte code format

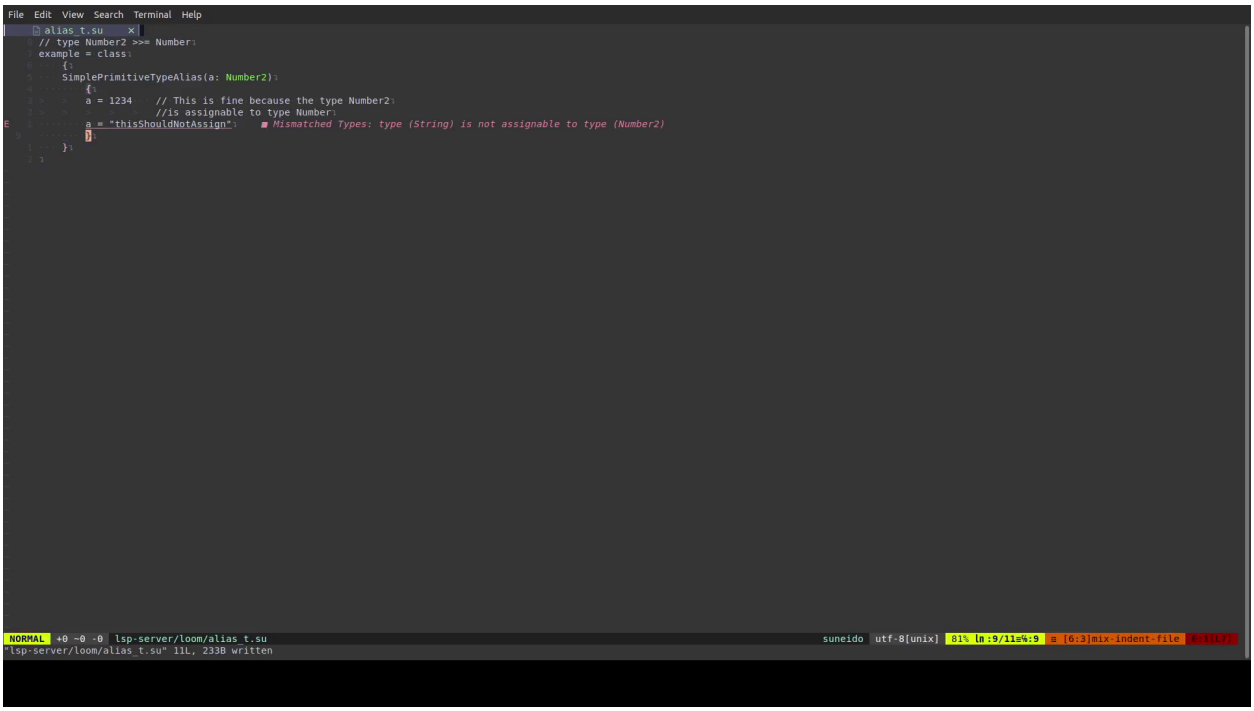

Figure 7: An example of type aliasing in TypeLoom

4 Demonstrations

4.1 Type Aliases

A common feature across gradually typed systems is type aliases. Type aliases are the first step toward allowing developers and users to define custom types or even build new types from already existing primitives. For example, a user can define a custom type *Message* which is nothing but the type *String*. In TypeLoom, we use the Haskell monad bind operator to define types. The syntax used to define the types is purely an aesthetic choice as the actual types would be taken in through code actions.

```
type Message >>= String
```

As demonstrated in 7 the parameter *a* to the method *SimplePrimitiveTypeAlias* is defined to be of the type *Number2*, also indicated by the comment on line 1. It should be noted that the comment only exists to provide context to the demonstration and is not actually used by the TypeLoom system. It is okay to assign the value 1234 to the variable *a*, as the value 1234 is compatible with the type *Number*, but in the next line when the variable is being assigned a value of the type *String* the system promptly throws a type error.

4.2 Union Types

A union type simply put allows a variable to have either of the multiple types defined therein. This type is especially useful for gradual types as it allows the developers to represent certain invalid states. For instance, a union type defined as the following:

```
type ID >>= Date | String
```

The type *ID* here is defined such that it maybe of type *Date* or of type *String* but, not both at the same time. This is particularly useful for null-able types. As demonstrated in 8 the user has defined a custom type *Currency* which has only three valid states, either the string literal "USD" or "CAD" or "GBP". Each of the variables *u*, *nu*, and *ou* are defined to be of type *Currency* and we see subsequent errors shown to the user when the values do not match what is expected of the defined type.

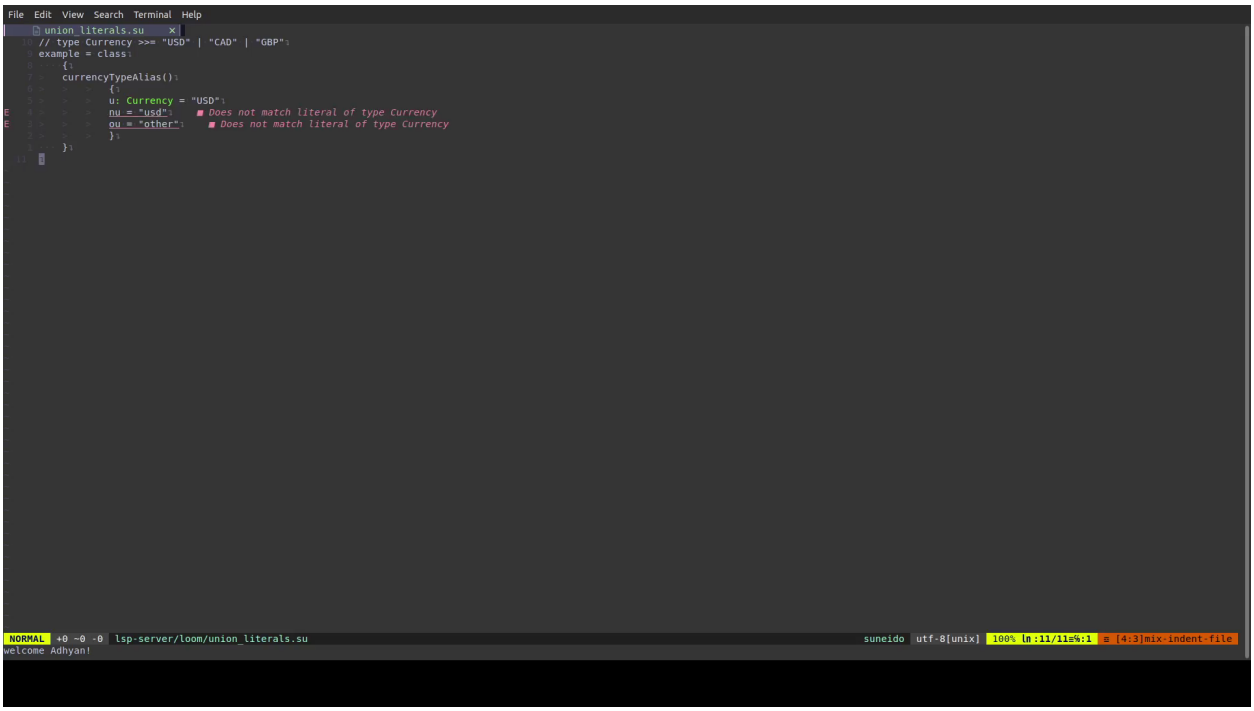

Figure 8: An example of union types in TypeLoom

4.3 Structural Types

Structural types are types whose compatibility or equality is determined by the definition of how the type is constructed. Structural types as the name suggests are useful in defining the type of particular objects like structs. Structural types are excellent for modelling the behavior of Suneido Objects.

In figure 9 we see two types that are of interest to us,

```
type User >>= Object(name: String, age: Number)
type Admin >>= User & Object(role: String)
```

The ampersand sign is used to define intersectional types. Intersectional types define a new type that must have a value that can be assigned both of two other given types. Intersection types are only supported on Objects as of now. In the type definition we can see a new type being defined which is type *User* which has two fields *name* of type *String* and *age* of type *Number*. Another type *Admin* is defined with the same two fields in the type *User* and an additional field *role* which is of type *String*. In the demonstration, two variables *user* and *user2* are inferred to be of type *Admin* and *User* respectively based on what the methods *NewUser()* and *NewUser2()* are annotated to return by the user. When we try to pass the variable *user2* whose type is *User* to a method *GetUserAuth* which expects type *Admin* we get an error as the type *User* cannot be passed in place of the type *Admin*, but it is safe to pass the type *Admin* where a type *User* is expected. It can be said that the type *Admin* is inherited from the type *User* or in more technical terms, *Admin* is subtype of *User*.

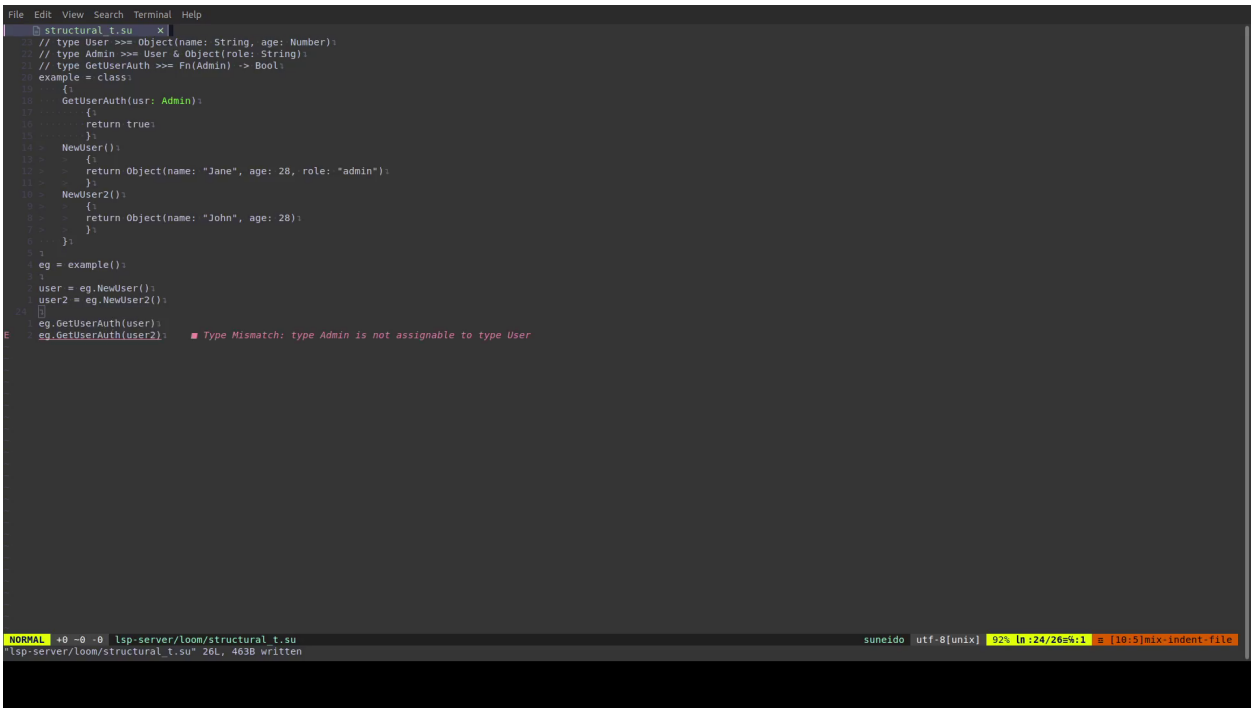

Figure 9: An example of structural types in TypeLoom

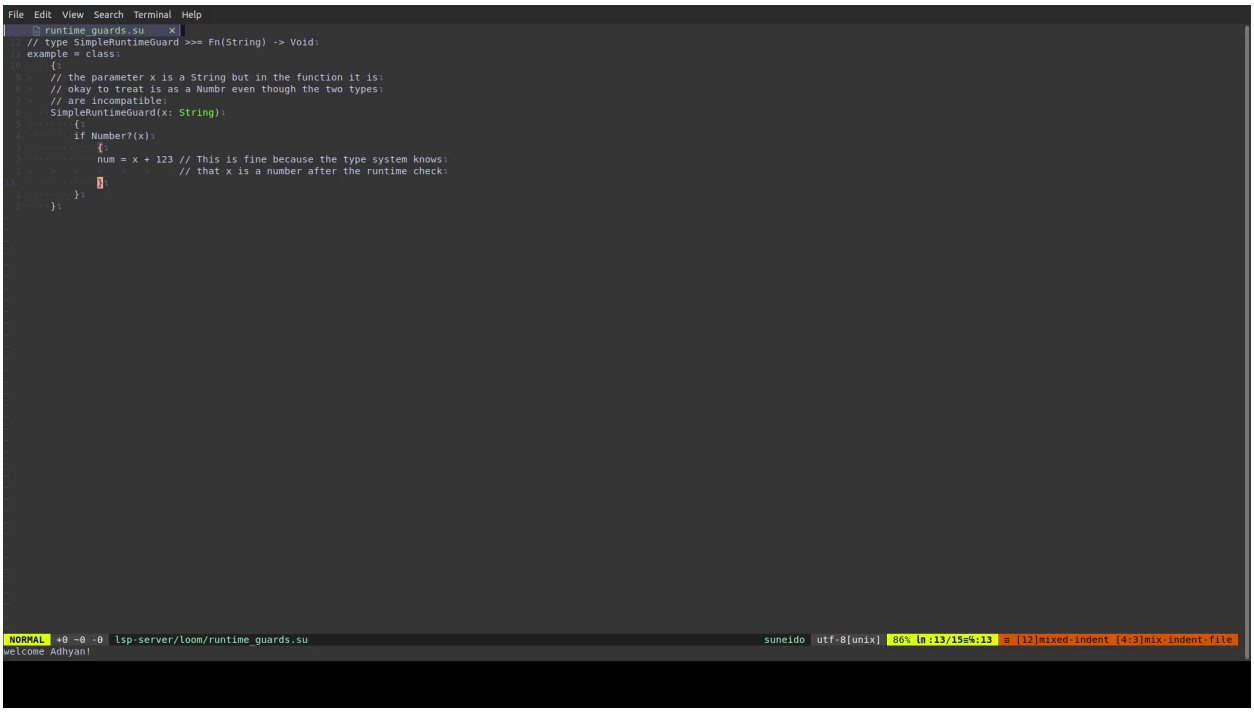

Figure 10: An example of type guards based inference and checking in TypeLoom

4.4 Type Guards based inference

Many languages that have dynamic typing, also support some level of type reflection i.e. the ability to examine the type of a variable at runtime. This is useful in asserting a variable is of a certain type. Suneido also supports this, and TypeLoom leverages this feature to gain additional information about what type a variable is. In 10 we see that even though the argument *x* is defined to be of type *String*, when it is used in an addition operation, which we have demonstrated before to only work for the *Number* type, out type system throws no such errors. This is because the if statement, checks if *x* is of type *Number*, and only proceeds if that is true. The code within the if-statement can be thought of as unreachable code as *x* will never be of type *Number* and always be of type *String* if TypeLoom does its job correctly.

5 Drawbacks

1. The system of using Code Actions to take in type information wasn't implemented due to time constraints.
2. The system successfully generated byte code but the VM that executes it isn't implemented so, for the purposes of the demonstrations, the execution of the byte code was hard-coded and not dynamic.
3. It is unclear if the approach of using a graph would scale on an enormous project.
4. We have not mathematically proved that this type system is correct.
5. I have observed the type system to be undecidable, as for certain combinations of type guards, it is unable to infer and check types. A rather simple solution to this problem would be to eliminate type guards based inference

6 Conclusion

In this paper, we introduced TypeLoom, an innovative tool designed to seamlessly integrate gradual, optional typing into legacy codebases of dynamically typed languages. Our approach leverages the Language Server Protocol (LSP) to provide real-time, in-editor type information through inlay hints and code actions, facilitating the collection and refinement of type data without the need for syntactical changes or specialized comments. By employing a graph-based data structure, TypeLoom enhances the flexibility and accuracy of type inference and type checking, making it particularly well-suited for gradual typing.

Through the implementation for the Suneido programming language, we demonstrated how TypeLoom effectively gathers type information, displays it to users, and checks for type consistency, all while adhering to the principles of gradual typing. Our solution addresses the common pitfalls of existing approaches by avoiding the need for new syntax, compiler changes, or syntactically correct comments, thereby providing a more intuitive and non-intrusive experience for developers.

The demonstrations showcased the practical applications of TypeLoom, including type aliases, union types, and structural types, highlighting the system's versatility and robustness. The ability to define custom types and detect type errors in real-time significantly improves code maintainability, readability, and developer confidence during refactoring processes.

In conclusion, TypeLoom ties together existing technologies and contributes a novel way of using them together to enhance develop experience when migrating large dynamic code bases to ones with types.

6.1 Future Work

- **Generics and Polymorphism:** Extending the system to infer and check types for generic data structures and polymorphic functions.
- **More Flexible Typing Rules:** Allowing developers more flexible rules on defining types and extending the system to suit more languages or even allow custom type checking rules.
- **Incremental Type Checking:** Implementing incremental type checking to only recheck parts of the code that have changed, reducing the overhead for large projects.
- **Recursive Data Structures:** As the system currently stands there is no easy way to represent a recursive data structure like nested lists.
- **Interactive Type Debugging:** Developing interactive tools for debugging type errors, allowing developers to explore the type graph and understand the propagation of type information.

- **Proof of Correctness:** Developing formal proofs of the correctness of the type inference and checking algorithms to increase confidence in their reliability.

References

[Gje21] Jon Gjengset. *Rust for rustaceans: Idiomatic programming for experienced developers*. en. San Francisco, CA: No Starch Press, 2021.

[Mel23] Mike Melandon. *TypeScript and the dawn of gradual types*. en. July 2023.

[Smi23] Ethan Smith. *Part 1: Bidirectional Constraint Generation*. https://thunderseethe.dev/posts/bidirectional-constraint-generation/. Accessed: 2024-6-8. June 2023.

[Erw] Martin Erwig. *Visual Type Inference*. https://web.engr.oregonstate.edu/~erwig/papers/VisualTypeInf_JVLC06.pdf. Accessed: 2024-6-8.

[McK] Andrew McKinley. *Contents*. https://suneido.com/info/suneidoc/Contents.htm. Accessed: 2024-6-8.

[Mic] Microsoft. *Specification*. en. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/. Accessed: 2024-6-8.

[SQL] SQLite. *Why SQLite Uses Bytecode*. https://sqlite.org/whybytecode.html. Accessed: 2024-6-8.