Zenith: Type Safe, Functional Programming Language for Lua

by

Abhash Shrestha

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

May 2023

ABSTRACT

This paper introduces Zenith, a statically typed, functional programming language that compiles to Lua modules. The goal of Zenith is to be used in tandem with Lua, as a secondary language, in which Lua developers can transition potentially unsound programs into Zenith instead. Here developers will be ensured a set of guarantees during compile time, which are provided through Zenith's language design and type system. This paper formulates the reasoning behind the design choices in Zenith, based on prior work. This paper also provides a basic understanding and intuitions on the Hindley-Milner type system used in Zenith, and the functional programming data types used to encode unsound functions. With these ideas combined, the paper concludes on how Zenith can provide soundness and runtime safety as a language, and how Zenith may be used with Lua to create safe systems.

ACKNOWLEDGMENTS

I would like to express my deep gratitude to my undergraduate chair and current co-chair Professor Gennaro De Luca, as well as the rest of the committee, for this opportunity and for their infinite patience, trust, and feedback throughout the course of this project. I thank you all not only for reading and providing feedback on this thesis, but for also teaching the classes that evoked these ideas long before this project began. I also thank my graduate advisors Wendy Biresch and Wendy Gibson-Wright for their invaluable guidance and assistance in pursuing this project, this project would likely not have been possible had it not been for their planning and insights.

Lastly, I would like to thank my family and friends for their encouragement and advice during this journey, I will always be inspired by you all.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Between statically and dynamically typed programming languages, dynamically typed languages are a popular choice among developers for short scripts and programs, thanks to the simplicity and low overhead in their language semantics. However, when these scripts develop into being larger programs and systems, bugs can quickly arise that may have not been apparent at smaller scales. These bugs oftentimes may have been easily caught by a simple static type checking system, and even more can be eliminated with more sophisticated design choices. Indeed, the static analysis that these type checking systems can provide are invaluable for reducing runtime errors before a program even begins. Consequently, a recent trend in programming language development has focused on providing type safe alternative languages, or typed supersets of dynamic languages, so that these dynamically typed programs can be extended safely.

As such, this paper introduces Zenith, a type safe, functional programming language that compiles to and interoperates with the Lua programming language, in the form of modules. The Lua language is a popular, dynamically typed language, that is often used in embedded contexts. With no static typing system however, it lacks safety guarantees that can be found in statically typed language. Zenith is designed to provide these type safety and runtime guarantees to Lua developers, while introducing minimal overhead to a Lua environment. Furthermore, since Zenith focuses on the functional programming paradigm, it utilizes Hindley-Milner (HM) type inference, which allows for easier developer usage and simpler semantics based on type inference, resulting in less-error prone programs. Ultimately, the goal of Zenith that is investigated in this paper, is to approach a total guarantee on runtime safety with modules compiled by Zenith.

To achieve this goal, this paper will lay out the fundamental groundwork behind Zenith's design choices in Chapter 2, where we will form an understanding of the prior work involved in runtime safety, static type systems, and functional programming. We will also review other notable attempts different programing languages make in providing safety guarantees. Furthermore, while this paper assumes some knowledge of functional programming, this paper is intended to be understandable for those not familiar in type theory and will therefore attempt to summarize and provide intuitions behind notable types, algorithms, and systems behind the HM type system and Zenith's. Once we have established an understanding of these programming language and type system aspects, we will begin to construct the language of Zenith in Chapter 3. Here, the paper will cover the design choices by Zenith, based on the knowledge gathered in Chapter 2. Finally, Section 4 will briefly cover the additional implementation details that were not included in Section 3, for the sake of reproducibility. With this all combined, we will see how Zenith is able to be a flexible, functional, and most notably safe programming language for Lua.

CHAPTER 2

BACKGROUND AND PRIOR WORK

The design and implementation of Zenith builds upon the prior work and commonly known aspects about type systems and safety, as well as functional programming. The goal of the language is to guarantee a safe runtime, while both compiling to a dynamic language, and being in used in tandem with the target language. As such, most aspects regarding implementation are not necessarily novel, but are still interesting and compelling in the combination of design choices that are used in the makeup of Zenith. This chapter primarily focuses upon the motivations and tradeoffs behind those choices. Furthermore, based on the existing state of the art, this chapter will also attempt to provide a brief understanding of the components considered for and implemented in the Zenith language. This includes the static typing and type safety features, as well as functional programming, functional data types and type systems, effects, and language interoperation. As part of a complete review, we will also briefly investigate the considered but ultimately rejected features for the Zenith language, being Turing incompleteness and totality, to understand the reasoning more fully behind these decisions.

2.1 Static Typing and Type Safety

Static typing is commonly known as the usage of a well-defined type system, such that the types of different expressions in a program are known to the compiler or type checker during compile time. This may be implemented in a language's semantics through the usage of static type annotations and type inference. With this, statically typed languages can eliminate incorrect programs before runtime even begins, as well as being able to include type level abstractions in the language semantics. Indeed, not only does a static type system check for compile time errors, but it also forces the programmer to consider potentially erroring components while writing the program. Other benefits of static typing that are not included in Zenith include deeper code optimizations, autocompletion, IDE tools, etc. While later chapters will explain more on static typing as it related to functional programming, we can see already how static typing systems are the basis of any programming language attempting to reduce runtime errors.

2.2 Functional Programming

Zenith's strict adherence to the functional programming paradigm is also an equally important aspect of the language to consider. In a functional programming language, functions are regarded as first-class, meaning that functions themselves can be passed and referenced in the same way as other bindings. Furthermore, functions may accept other functions as a parameter, in what is called a *higher-order function.*

$$\mathrm{fn} : \forall\alpha\forall\beta(\alpha \rightarrow \beta) \tag{1}$$

$$\mathrm{map} : \forall\alpha\forall\beta\big((\alpha \rightarrow \beta) \rightarrow (\alpha list) \rightarrow \beta\mathrm{list}\big)$$

As we can see in (1), map is a higher-order function, that may take a function like fn as an argument, so long as fn matches the types necessitated by map. This builds up many different design patterns, and as such, functional programs are often the composition of many functions being brought together in this manner.

However, this trait is not entirely unique to functional languages, and many multi-paradigm or even object-oriented languages include some implementation of higher-order functions, such as lambdas in Python or Java. What sets functional programming further apart, is that functional languages follow certain self-imposed constraints, that in exchange provide for corresponding guarantees. These constraints are that functions must be *pure*, and that all values are *immutable*. Functions being pure in this context denotes that functions should always maintain the same output give the same set of inputs. As an example, consider a function like $sine$, where it will always return the same number value if given the same input. Conversely, a random number generation function (without any seed value), may be considered as impure, as the return value cannot be predicted based on input. With the second constraint, values being immutable is to say that all values in the program cannot change implicitly throughout the runtime. After being declared, the value of an identifier is set for its declared scope, and the programmer can rely on that certainty. For simple primitive type values, this means assigning a new binding for when a number or Boolean for example is calculated, rather than mutating the original binding. More complex data types may be copied with new changes when an update is required, without modifying the original reference, nor the underlying value or values. Immutability provides a level of clarity and transparency that mutability may not. In a mutable program, a variable may make a surprising change after being passed through a function, which can lead to unexpected behavior and errors.

The combination of these constraints allows for programs to be deterministic, where both the programmer and compiler itself can expect guaranteed outcomes based on inputs. Nonfunctional, statically typed programming languages may include strong type systems like Zenith, but the lack of these guarantees is often a shortcoming in protecting runtime type safety. For instance, in these languages, even after typing checking an object type at compile time, behavior during runtime may mutate the object structure, which can cause expected properties of the object to change, causing errors when interacting with the object.

2.2.1 Hindley-Milner Type Systems

As seen in certain functional languages, like Clojure, Elixir, or Erlang, functional programming languages do not need to be statically typed, and indeed the properties of functional programming do not necessitate any such type system. Similarly, an inverse is also true; many statically typed languages are imperative or object oriented. However, when using the two language properties in conjunction, as in Zenith, or other languages like Haskell (Jones, 2002) and Elm (Czaplicki, 2012), a language can gain interesting and useful additional capabilities in this intersection.

To begin, many of these statically typed functional programming languages follow the type system known as the Hindley-Milner (HM) type system. HM is the groundwork for many of these successful statically typed functional languages, thanks to the variety of features in its framework that we can see be elaborated in Mark P. Jones' paper *Functional Programming with Overloading and Higher-Order Polymorphism*, being type security, type inference, flexibility, and ease of implementation (Jones, 1995).

Type security, or soundness, in HM according to Jones, ensures that the behavior of a type system is consistent and guaranteed based on the types of a well-typed program. We can expect a language that has properly implemented HM to have correct values for bindings based on the type of binding. Furthermore, the type inference that HM provides is a convenient tool for writing concise programs, without littering the entire syntax with type annotations and operators. Programmers who do still choose to use additional type annotations also benefit from type inference applying a consistency check to their code against the declared type annotation. Regarding type security however, HM is expected to make conservative assumptions for inference based on annotated bindings. What is notable here is that we must be sure that the language using the HM system does not incorrectly infer a type and will prioritize caution over an unsafe inference.

Flexibility in this context refers to the property of polymorphism in HM, which enables much of the type level abstractions and safety that Zenith and established HM-based languages rely on. Polymorphism allows function types and kinds to generalize and abstract over multiple data types, represented often by a type variable. This contrasts with monomorphically typed languages, where a function may need to be rewritten in several implementations to operate on arguments and return values of different types (Jones, 1995). We will see the usage of these polymorphic functions in the next section covering functional data types, where polymorphism will assist these abstract data types in covering a wider range of types more concisely.

2.2.2 Functional Data Types and Type Classes

Functional programming languages are generally split into two different methodologies, beyond static and dynamic typing. These are languages that (attempt to) adhere to the purity constraint of functional programming, like Haskell and Elm, and languages that do not, such as Scheme and Erlang. Languages that do not are allowed to perform an arbitrary degree of different side effects, such as IO. This may make code easier to write at first but can introduce unexpected behavior when calling and composing functions. On the other hand, of course, most “pure” functional languages often still need to perform some degree of side effects and IO to be observable and do anything of practical use. However, designers of these languages are still able to eliminate much of the actual impact of performing effects on the language semantics, through the usage of different functional data types and type classes (Jones, 2007).

Pure functional languages based on HM and other similar type systems often make use of a similar set of data types, implemented on polymorphic type classes. Type classes in functional programming define a set of functions that can be performed for a particular type (Wadler & Blott, 1989). The fundamental type classes discussed in this paper are often rooted in category theory, dealing with the composition of different structures, a practice very familiar to functional programmers. However, for brevity, this paper will forego the detailed category theory concepts behind each type class and data type and will focus on the practical applications of pertinent structures regarding functional programming. It should also be noted that the following types may not necessitate type classes for their implementation, but assuming the semantics of type classes can make the discussion of their general use more straightforward.

To gain a baseline understanding of the data types relevant to runtime safety, we begin with a breakdown of the essential functional type classes. To build an understanding of functional data types, we begin with *functors.* Functors are an essential component of many statically typed functional languages. Functional programmers will be familiar with the map function, which can be derived from the $fmap$ function used with the functor type class. Here, the functor type class represents data structures that possess some value that can be mapped to a different value of the same structure. For instance, functional languages often consider the common data type $List$, as a functor. Here, $List$ α would the type for a list of type α, and $fmap$: $(\alpha \rightarrow \beta) \rightarrow \mathrm{f}\ \alpha \rightarrow \beta$, the mapping function, would be how that functor $List$ α may be mapped to a $List$ β, after the function is applied with an expression like $fmap\ \big(\lambda x.x{:}\ \alpha\ (\mathrm{x} \rightarrow \beta)\big)\ (List\ \alpha)$. In the context of this paper, the primary aspect of functors that we take away is how we might "transform" and encode an underlying type, based on the structure of a type

Monads are a similar data type used in functional languages. To explain, it may be simpler to begin with types of the necessary functions in a monad (Wadler, 1995).

$$\mathrm{Unit}{:}\ \forall\alpha(\alpha \rightarrow M\ \alpha) \qquad (2)$$

$$\mathrm{Bind}:\ \forall\alpha\forall\beta(\mathrm{M}\alpha \rightarrow (\alpha \rightarrow \mathrm{M}\beta) \rightarrow \mathrm{M}\beta)$$

It is also worth noting that the names of these functions may differ in implementations, such as unit being often referred to as $return$ in Haskell, and bind being the operator $>>=$ as syntax sugar. However, the function types should remain the same between different implementations. Regardless, here the $unit$ function can be understood as a function that lifts a value into monadic form. For example, for a $List$ monad, the value of $unit$ α would return $List$ α. This is useful to combine with other monadic functions later.

The $bind$ function, as we can see from the type signature, is a higher order function that takes a monad of value α, and a function which takes underlying value α to return value β, and returns a monad β. In this case, it may help to think of the bind function almost as a combination of $concatenate$ and map. Returning to our $List$

example, if we choose to bind a $List\ Int$ of $[1,2,3]$ with a function converting Int to $[Char]$, we may receive the value "123", equivalent for $['1','2','3']$, being each value mapped to a $Char$ type, and concatenated together into a $List\ Char$. Contrast this with the $fmap$ function of functors. If we were to use map in place of bind here, we would end up with the value $[['1'],['2'],['3']]$, the type $List\ List\ Char$. Essentially, monads have enabled us to write a composition between a structure, and a function that "contributes" to the structure. While the benefits to monads here may seem abstract and unapparent, we will cover them more in depth in Section 2.2.3.

We cover these data types because of their usage in statically typed functional languages to model different structures and behaviors. Particularly, we are interested in how these types abstract over the effects and exception behaviors, to see how they may be used in providing soundness. Fundamentally, as we have seen here, by abstracting certain behaviors into the type of a data structure, we can enforce the proper usage and handling of said behaviors.

2.2.3 Effects

With the described properties and constraints of functional programming as mentioned, an issue that arises is how a functional language may model the usage of effects, such as IO, and in the case of Zenith, exceptions. We briefly established the foundations for the functional data types that may be used to model such structures, so we will begin to deconstruct in what manner effects may modeled, such that the semantics of the language remain "pure", and so that we can provide additional safety to Zenith.

In a paper by developers of the Haskell language, *A History of Haskell: Being Lazy with Class,* S.P. Jones. describes the progression of Haskell's effect handling methods (2007). We see that Haskell eventually decides on the usage of monads to model effects in a publication by Philip Wadler, *Monads for functional programming* (1992). Using our understanding of monads and functional data types, we see here that the Haskell Committee decide to model a "computation" of type α, under monadic form, as the M α type we are familiar with. Furthermore, as shown in the paper, we can compose additional functions onto the result of the computation, using the $bind$ function. For example, with our computation monad M α, the expression $m \gg= (\backslash x \rightarrow n)$ has the type M β, where β is the resulting type of the higher order function. In an imperative language, or even in a functional language using let bindings, this "reassignment" of the computation result value would not be made apparent in the typing of the program, and the effect of computation would not be visible.

What this means for a strong type system, and for runtime safety, is that that encoding values this way makes these otherwise "invisible" effects, like IO, or exceptions as an Either monad, very apparent in the type signatures of our functions. Furthermore, given this ability to abstract the existence of effects into our type system, we can ensure in the semantics of a language that the results of these effects are never ignored by a program writer, by designing any potentially unsafe standard library function with this model in mind. Indeed, languages such as Elm, and of course Haskell as we observed, provide type and runtime safety through this means.

However, there is still some degree to which runtime safety can be further improved, which a few select languages attempt to make. For example, in addition to Elm, the Dhall programming and configuration language all but guarantees runtime safety. As a brief overview, Dhall is a Turing incomplete language, with a syntax like that of Haskell's. Dhall is interestingly specific and concise in its goals, which is to act as a programmable, but also relatively inert, configuration language. To achieve this, it completely forbids any side-effects. Note that this is significantly different than abstracting side-effects with the use of an IO monad. This is in accordance with Dhall's stated "marketing" of displacing YAML. It deems that it would be undesirable for a configuration language to perform arbitrary effects when referencing data from it. Furthermore, Dhall's Turing incompleteness *generally* provides a guarantee that the program will complete in reasonable time, which is another aspect that developers would expect from a configuration language. While Dhall's Turing incompleteness will be discussed in later sections, these aspects still provide a novel intuition on approaches to runtime safety. Primarily, Dhall's usage as a "second language" of sorts allows it to impose constraints on itself and offer according guarantees. Developers using a less safe language can migrate aspects of their code configurations to Dhall for more soundness and rely on the safety of Dhall instead ("The Dhall configuration language"). This aspect will be the greatest influence of Dhall on the design of Zenith, which we will see in Chapter 3.

Focusing back on Elm, despite being somewhat more of a feature heavy language, Elm still attempts to provide a guarantee on throwing no runtime errors ("A delightful language…"). While this is not always true in any arbitrary edge-case program, the remaining types of runtime errors are relatively uncommon occurrences, and not something we necessarily consider in a formal language (Payr, 2021, p. 17). These exceptions include out of memory errors, function comparison, and infinite recursion. While these exceptions can be significant in select programs, we will not focus on out of memory errors, and for the sake of brevity will forgo allowing function comparison to take place in our language semantics. We will cover issues with recursion in brief in Section 2.3.

Elm follows a similar doctrine as Dhall. It has a concise “market”, in Dhall’s terms, to be a language for front-end webpages. As such, it is relatively difficult in Elm to perform arbitrary side-effects outside of this target domain, and effects within are safely abstracted. Of course, being a Turing complete language, it can be used for different programs, but pragmatically, within the language semantics, it is designed to implement those specific set of programs. This allows Elm to reduce the uncertainty involved in performing IO, and the potential for “silently erroring” during execution. Elm in this sense is a concise second language, designed to interact with other languages when developing a more interactive website.

As stated before, realistically, Elm and Dhall can still encounter runtime errors. But they are still novel and state-of-the-art languages in their effort of preventing them, as we have seen in their design. Given the similarity between the high-level type systems employed between Elm, Dhall, and other languages like Haskell, we can understand that there are still more aspects about a language’s semantics and implementation that can help prevent runtime errors. In this case, an interesting and novel aspect of Elm and Dhall that helps them provide more guarantees, is the constraint that they impose upon themselves on what programs it can compile.

We will further reason about maintaining a strict language domain in Section 3, where this paper will cover how Zenith follows a similar principle in its semantics to tactfully exchange responsibility of execution with Lua. In summary however, the functional data types that are offered by strong type checking system, as well as functional purity, allow us to abstract the potentially erroneous effects and exceptions of a program, into a model that can be deconstructed using strongly typed functions. These models protect the "edges" of our program that interact with unsafe data and allow us to achieve greater runtime safety. In chapter 3, we will see how these data types take form in Zenith and its standard library and see how Zenith is able to protect itself against errors through these language semantics.

2.3 Totality and Turing Completeness

As an aside to type checking, it is also worth investigating total programming, and the value, or lack thereof, of Turing completeness and what tradeoffs can be made. Originally in this research project, considerations were made for Zenith to be a Turing incomplete language. Turing incomplete languages are generally able to provide a greater degree of verifiability before runtime. This is due to the property of Turing incomplete languages being able to "subvert" Rice's Theorem.

Rice's theorem (Rice, 1953) essentially states that nontrivial properties of languages are undecidable (Kozen, 1977). As such, this implies that static analysis of Turing complete languages is limited to some degree, or is otherwise undecidable as well (Pickard, 2020). The idea behind many Turing incomplete languages then is to avoid this problem all together by forbidding the possibility of the halting problem, which Rice's theorem proves to be a reduction to. This includes forbidding arbitrary recursion, by banning it all together, or relying on means such as dependent typing to prove termination after recursion. As we saw in Dhall, the language takes the approach of forbidding recursion to achieve Turing incompleteness, guaranteeing its own termination.

However, it is known that while these languages forbid an infinitely long runtime, they cannot forbid an arbitrarily long one. This is not necessarily a massive flaw in their design; of course, implementors of these languages are aware of such an issue, but this speaks more to the aspect of "marketing" in their languages, in Dhall's terms. We argue that including Turing incompleteness in a language's design is a signal or summary of the goals of a language to potential users, but in of itself is a relatively inconsequential aspect, excluding dependent typing. A language designer who builds a language that is type safe and Turing incomplete will be aware and cautious of other safety issues in a language's semantics, which is generally what appeals to developers looking for a safe language. Indeed, the communities behind Turing incomplete languages like Dhall and Idris (Brady, 2013) place a heavy emphasis on safety and soundness. However, in the absence of dependent typing and other features like hashing in Dhall, we find that Turing-incompleteness is not particularly useful for a language like Zenith. The only true benefit to Turing-incompleteness for a basic HM system, appears to be guaranteeing termination. This is somewhat inconsequential when considering that termination is not itself a significant guarantee, as it can still allow programs to run for an arbitrarily unlimited amount of time, which just happens to be predefined. We would also be able to introspect more into programs before runtime given Turing-incompleteness, but given the exclusion of side-effects already and usage of HM over dependent typing, there is not much benefit for this introspection.

In summary, we have researched the various aspects that we will include or exclude from Zenith, such that Zenith may implement programs that are guaranteed to be safe during runtime. We primarily investigated how functional languages, and particularly statically typed, HM languages, protect their runtime by catching errors during compile time, or force them out of the language semantics all-together. With this foundational knowledge, we can begin to construct the language design of Zenith formally.

CHAPTER 3

THE ZENITH LANGUAGE

The language of Zenith, as mentioned prior, greatly resembles the Haskell language, albeit with simplified semantics. This was chosen due to Haskell's predominant role in the functional programming space, and general recognizability to functional programmers. Furthermore, Haskell's declaration and module semantics provide a straightforward mapping to Lua's module system, because of their relative similarity in certain use cases, which we can dictate. It also provides a strong foundation for the implementation of the HM type system that is intended for Zenith.

Section 3.1 will provide the Zenith syntax in Backus-Naur-Form. Section 3.2 will cover the type system for Zenith, the implementation of HM and typing judgments made, and the implementation of Algorithm W of HM. Section 3.3 will showcase design of the Zenith standard library for functional programming and type safety. Finally, Section 3.4 will cover the considerations of Zenith regarding IO, and how Zenith interacts with Lua as a second language.

Zenith's syntax is as follows:

$$
\begin{aligned}
program &::= module \\
module &::= header\ body \\
header &::= \text{"module" id "where"} \\
body &::= declaration+ \\
declaration &::= signature\ implementation \\
signature &::= id\ \text{"::"}\ typeDeclaration \\
keyword &::= \text{"}if\text{"} \mid \text{"}then\text{"} \mid \text{"}else\text{"} \mid \text{"}data\text{"} \\
binFn &::= \text{"}+\text{"} \mid \text{"}-\text{"} \mid \text{"}*\text{"} \mid \text{"}/\text{"} \\
binFnApp &::= binFn\ expr \\
boolean &::= \text{"}True\text{"} \mid \text{"}False\text{"} \\
id &::= -keyword\ idStart\ idPart* \\
idStart &::= letter \mid \text{"\_"} \\
idPart &::= alnum \mid \text{"\_"} \\
csv &::= \text{","}\ expr \\
list &::= \text{"["}\ expr\ csv*\ \text{"]"} \\
atom &::= digit+ \mid str \mid boolean \mid id \mid list \\
expr &::= startExpr\ endExpr \\
startExpr &::= atom \\
endExpr &::= binFnApp \mid expr \mid \epsilon \\
typeDec &::= type\ followingTypes* \\
followingTypes &::= \text{"}\rightarrow\text{" type} \\
type &::= basicType \mid listType \\
basicType &::= upper\ idPart* \\
listType &::= \text{"["}\ basicType\ \text{"]"} \\
impl &::= id\ args\ \text{"="}\ expr \\
args &::= identifier*
\end{aligned}
\tag{3}
$$

In implementation, the grammar of the language is slightly more complex and has several lower-level definitions, as Zenith does not include a lexing stage. Notably, we have excluded rules for parsing whitespace. Additionally, it may be noticed how expressions are split into a start and end expression form, where the starting atom is parsed, followed by various possibilities of the end expression, such as function application, or binary function application. This is to assist the recursive descent parser by factoring out left-recursive grammars, which would otherwise not terminate.

3.2 Type System

Zenith's complete type system is an implementation of type inference following HM, and type assignment using Algorithm W of HM, as described in *Principal Type-Schemes for Functional Programs* by Luis Damas and Robin Milner (1982). This section will provide some brief introduction to understanding any type notation used.

After simplification, Zenith's core language essentially becomes a collection of a function definitions. A function is defined in two parts, through an (optional) type annotation, and through the value definition, which is the name of the function, followed by the declared bindings for arguments, and then by the body expression. To begin, we will look at the core language of expressions, which is almost identical to that given by Damas and Milner in the first definition of Algorithm W (1982) and to the lambda calculus, with the exclusion of let expressions. Here we define expressions as e and the set of identifiers as x.

$$\mathrm{e} ::= \mathrm{x} \mid \mathrm{e}\ \mathrm{e}' \mid \lambda \mathrm{x}.\, \mathrm{e} \qquad (4)$$

With the language in (4), we represent the core functionality of expressions in Zenith. For example, more complex expressions, like if expressions, are reduced to this form. Now, the first step in an HM type system, is for it to deduce a *type-scheme*, in other words a polymorphic type inference, based on some string from this language. In general, all types in HM are constructions of constant types like *int* and *bool*, or type variables like α, β, which are combined using type operators. Based on the expression language, we infer type-schemes based on this logic, which will give us a similar form of $types$ τ, and of $\text{type} - \text{schemes}$ σ as the type-schemes paper (Damas & Milner, 1982).

$$\tau ::= \alpha \mid \iota \mid \tau \rightarrow \tau \qquad (5)$$

$$\sigma ::= \tau \mid \forall \alpha \sigma$$

In (5), we can see constant, primitive types represented as ι, type variables as α, and a function type as $\tau \rightarrow \tau$. Type schemes map to either a direct $type$ τ or a quantified type scheme, for a type variable α. This is the standard makeup of our language for types. From here, we collect a set of assumptions about expressions and type-schemes, written as:

$$A \vdash e : \sigma \qquad (6)$$

This can be read as "given a set of assumptions A, it can be proven that e is of type-scheme σ". The set A here is derived from the standard type inference rules that Damas and Milner uses (1982), which are rewritten below for our understanding:

$$\text{TAUT: } \frac{}{A \vdash e : \sigma} \qquad \text{INST: } \frac{A \vdash e : \sigma}{A \vdash e : \sigma} \tag{7}$$

$$\text{GEN: } \frac{A \vdash e : \sigma}{A \vdash e : \forall\alpha\sigma} \qquad APP: \frac{A \vdash e{:}\,\tau' \to \tau \quad A \vdash e'{:}\,\tau'}{A \vdash (e\ e'){:}\,\tau}$$

$$ABS: \frac{A_x \cup \{x{:}\,\tau'\} \vdash e{:}\,\tau}{A \vdash (\lambda x.e){:}\,\tau' \to \tau}$$

We exclude the let inference used by the original HM implementation, as in Zenith let expressions are not a fundamental expression. Regardless, from these inference rules, the earlier set A of assumptions can be derived from arbitrary expressions, by composing sound inference rules together. In Zenith, the implementation of type inference is essentially just the translation of these rules to code form.

The final inclusion in the type system of Zenith is the implementation of Algorithm W provided by Damas and Milner (1982). Algorithm W allows our type system to find a type-scheme σ, such that $A \vdash e{:}\,\sigma$, given A and e. Furthermore, the algorithm will also find a substitution S and type τ, such that $SA \vdash e : \tau$, where applying S to assumptions A allows e to be a concrete type τ. To implement the algorithm, we first follow the unification algorithm U by Robinson in *A machine-oriented logic based on the resolution principle* (Robinson, 1965). However, we will still primarily follow the description of the algorithm used by Damas and Milner. Here the algorithm, given a pair of types τ, τ', either returns a substitution V which unifies $\tau\ \tau'$, or if a substitution S unifies τ, τ', returns a substitution R, such that composing the substitutions together as RV, equals S.

After this, the final Algorithm W is as so (Damas & Milner, 1982):

$W(A, e) = (S, \tau)$ *where the following conditions must be met based on the form of* e:

(i) If e is an identifier expression x, and x is included in the set of assumptions A, then substitution S will be the empty substitution, returning the same value of A, and τ will be the type bound by x.

(ii) If e is the application expression $e^1\ e^2$, then recursively solve $W(A, e^1) = (S^1, \tau^1)$, for the proper substitutions and types of e^1. Then recursively solve $W(S^1 A, e^2) = (S^2, \tau^2)$ for return type of the expression after application of the substitution. Finally, let β be a new type variable in the unification $U(S^1\tau^1, \tau^2 \rightarrow \beta) = V$, and return the final substitution as the composition of substitutions $S = VS^2S^1$ and the final type as the return type substitution on the type variable β, as $\tau = V\beta$.[1]

(iii) If e is the abstraction expression $\lambda x. e^1$, then let β represent a new type variable, and $W(A_x \cup \{x : \beta\},\ e^1) = (S^1, \tau^1)$, resulting in $S = S^1$ and $\tau = S^1\beta \rightarrow \tau^1$. This effectively binds x in the lambda expression to type variable β and provides the substitutions to constrain β.

Again, in our implementation we exclude the case of let expressions. Additionally, an improperly formed expression e will result in a compile-time error. Cases like the identifier expression e being unbound will also throw a compile-time error. In summary, by following the implementation of HM type inference and the type assignment algorithm W, we build the foundation of a strong and concise static type checking system

[1] In original publication of (Damas, 1982), there is a typographic error in part (ii). I have chosen to correct this error in this implementation of W, but correctness of this fix can be contested.

in Zenith. The rest of the work in type checking is trivial, and mainly involves reducing expressions to sound forms in HM. In actual implementation, much of the code also follows *Typing Haskell in Haskell* by Mark P. Jones (2000) for a Haskell translation of these algorithms. Regardless, with the HM type system in place, Zenith can provide the additional components of runtime safety that we researched in the prior review, namely functional data types and classes, which will be covered in the next section.

3.3 Functional Programming in Zenith

As discussed in the prior research, functional programming is essential in the pursuit of maintain runtime safety. Much of Zenith's standard library includes general purpose functional programming functions. This includes transformers like map, $filter$, $fold$, various composition operators, and functions to use with type classes like the Maybe type class. These functions allow for both the manipulation of functional control flow in place of imperative constructs, as well as for the transformation of immutable data. This is also a sort of byproduct and benefit to using Zenith that Lua developers may notice, where Zenith provides additional standard library components.

Being a functional language, Zenith excludes semantics that violate functional programming practices, regardless of whether they exist in Lua. Indeed, Zenith primarily focuses on its own semantics and goals, before including Lua features. For example, mutable variables are not possible in Zenith, as the language offers no such ability to re-assign a variable. Most notably, Zenith also enforces purity using the various methods studied in Section 2. For example, possible exceptions are always wrapped in a Result type, where a return value is either encoded as the successful value, or an erroring type.

With this, regardless of outcome, the result of the function must be considered at compile-time, which eliminates the possible of an impure return from functions that may error. Furthermore, null values are wrapped in a Maybe type, forcing the programmer to consider nullable values while writing the program, averting the risk of an error appearing during runtime. These functional data types and similar ones avert most runtime errors, but similar implementations for IO can still often error or result in unpredictable behavior, due to the interaction with external systems. IO monads, while a being a significantly useful tool in managing effects, do not fix the fundamental uncertainty of IO, and as such are excluded from Zenith itself. This will be explored further in the next section.

3.4 IO and Modules

Zenith is designed to work in tandem with the Lua programming language. In fact, Zenith on its own is not capable of many significant systems, as language semantics intentionally forgo IO. As we have seen from other runtime safe languages like Dhall and Elm, by constraining the codomain of a language, we are able to provide safer bounds on what the language may error upon. Particularly, Elm and Dhall restricting their own behavior offers more runtime safety than even the IO monad can. While Elm does have monad like operators, and a monadic Command type for IO, these are still generally performing effects within Elm's abstraction of webpages, like HTTP requests or DOM updates. Elm's Command type does not provide a means of executing other IO like accessing local file systems or launching other programs. Furthermore, as we saw in

Dhall, the language completely excludes arbitrary effects and instead operates as what we saw as a “second language”.

Indeed, this is the same principle guiding Zenith’s approach to effects, where we choose to exclude side-effects and offload the action of performing IO to the main Lua program using Zenith. While of course this may result in the greater Lua program being unsafe, what this allows for is for much of a Lua-Zenith system to be contained in a safe Zenith “interior”. Developers will know based on the guarantees provided by Zenith’s implementation that modules and libraries generated by Zenith are safe to use in runtime and can therefore offload more error-prone systems into Zenith. The responsibility of Lua in this case becomes simplified, and where the developer needs only to implement the imperative, IO “shell” of the program in Lua. Giving the uncertainty and flexibility that IO may introduce, this is a task already well-suited for dynamic languages and Lua, so long as we do not perform computations on this data before passing it into a Zenith-generated library.

By essentially wrapping much of the Lua language, Zenith is also capable of providing additional benefits to the Lua developer. As mentioned, Zenith builds upon and extends the Lua standard library with its own. Furthermore, Zenith can ideally abstract the somewhat fragmented versions of Lua into a centralized form. For example, Lua tends to be split between the mainline versions of Lua, such as Lua versions between Lua 5.1 to Lua 5.4, and LuaJIT, a fast, Just-In-Time compiler version of Lua. Converting code between these versions may not always be trivial, due to occasional syntax and semantic differences, but as an overhead layer, Zenith should be capable of compiling to different Lua versions. It should be noted however that the initial release of Zenith with this paper does not include this feature, but we still choose to elaborate feature due to it being a relatively common feature of superset languages like Zenith, such as Typescript. As such it is intended to be a future implementation.

As an aside, Lua includes many notable metaprogramming capabilities, such as metatables that allow developers to rewrite the semantics behind tables and objects. While certain aspects of metatables may be considered as mutations and non-declarative, unsafe code, there may be interesting aspects of metatables to include in Zenith. However, given the scope of this thesis, this is not explored in this paper.

With these features of Zenith combined, we have provided a rigorously safe, functionally pure language that eliminates runtime errors. Zenith provides a strong type system based on the well-established HM type system, encodes functions to safely handle and manipulate otherwise potentially erroring values in a functional programming manner, and integrates with Lua to handle effects and provide guarantees to users of the dynamically typed language. While edge case conditions exist for Zenith, as they do for other runtime safe languages, such as memory and resource errors, based on current state-of-the-art and other attempts at reducing runtime errors from a language semantics perspective, Zenith is a reasonable and sound approach to providing safety.

CHAPTER 4

IMPLEMENTING ZENITH

This section will briefly cover the programming implementation of the Zenith compiler. To begin, Zenith is implemented in Haskell. Given their similarity in syntax and types, this was a reasonable and useful choice in implementation, particularly for managing the type system. Zenith is parsed using parser combinators, a common function method of parsing languages. This has some consequences on the language grammar, as parser combinators are restricted in the grammars they can read. This includes left-recursive grammars, which are subsequently factored out of Zenith's lower-level syntax language. Furthermore, given the usage of parser combinators, the performance of Zenith's parsing can be exponentially long, but since this was not a major focus in this thesis, this issue is not addressed. After parsing, data types are coalesced into the AST data type.

The parser itself may parse forms that are syntax sugar for more fundamental forms, so from the AST, the next step of Zenith is to desugar and simplify various forms, such that they may be accepted by the HM type system. Particularly, functions of multiple arguments are reduced to a *curried* form, where each argument is applied individually, returning the function that accepts the next argument. This is generally standard behavior for functional language and is also the form that the type system reads for simplicity. Furthermore, because of the reliance on Lua, the language can circumvent certain expressions and reimplement them in Lua. For example, if expressions are reduced to $\forall\alpha\ cond(Boolean, \alpha, \alpha)$, where $cond$ is a replacement for if expressions included in the standard library. This also allows us to be sure that forms we parse as expressions indeed behave as expressions in the final Lua rendering, rather than an imperative statement.

The type checking system was already mostly covered in Section 3, but we will also briefly cover additional implementation details not specified by the HM implementation. Mainly, the HM implementation primarily focuses on the type inference and assignment of individual and nested expressions. In our language, we of course need to extend this behavior to the top-level function definitions and bindings. As such, the implementation maintains a single rolling environment of type-schemes and bindings for a module, and sequentially adds each binding and its type-scheme to the environment. This way, later calls to a top-level function are not unbound. Furthermore, the type checker must also check whether the type annotation of a function and the type inference of its implementation match, which we add in this implementation by running inference on implementation and parsing the annotation into a usable type to compare against.

Finally, given the corrected AST, the Zenith renderer will generate and format the proper strings of Lua code based on the AST. The renderer keeps a small degree of state between expressions, to maintain indentation, but does not include some notable features a renderer may have due to the scope of this paper. Namely, indentation is the only changing variable in the rendering process, and as such the rendering of functions is generally one-to-one between expression and Lua string, regardless of context. This means that the generated string is not formatted based on other expressions that may appear, which may not adhere to common Lua style guides. This also means that line-numbers may not necessarily match with the source Zenith code. Also, since Lua is of course dynamically typed, all type-related syntax is excluded during the rendering process. Finally, comments and certain whitespace choices by the writer are not retained during the rendering process.

These choices were mostly made due to the scope of this thesis and may be included in a more feature complete release of Zenith. However, despite these shortcomings, the implementation of Zenith is still generally robust, and adheres to the choices made in the language design.

CHAPTER 5

USAGE AND EVALUATION

As a demonstration of the capabilities of Zenith, as well as an overview of the workflow using the language, we will briefly cover examples of two libraries developed using Zenith. In total, these programs should cover the basic features of Zenith, as well as components of the standard library. To begin with a smaller program, we first look at the following Zenith code:

Figure 1

Code of Zenith Syntax and Types

```
addMaybe :: Number -> Maybe Number -> Maybe Number
addMaybe x my = mapMaybe (\y -> x + y) my

unsafe :: Number -> Maybe Number
unsafe n = n / 0

test1 = addMaybe 1 (Just 2)
test2 = addMaybe 1 (Nothing)
test3 = addMaybe 1 (unsafe 1)
```

In this code example, despite being a somewhat trivial program, we see how Zenith may be used to abstract and compose unsafe functions and other functions together. Zenith by default returns a $Maybe\ Number$ type for division, meaning that we do not encounter any unexpected behavior when dividing by 0. Being a $Maybe$ type, we can then map the underlying value further, and derive new values through this way.

Figure 2

Code of List Operations and Recursion in Zenith

```
-- Fibonacci Number

fib :: Number -> Number

fib 0 = 0

fib 1 = 1

fib n = (fib (n - 1)) + (fib (n - 2))


-- List Operations

concat :: List a -> List a -> List a

concat Nil bs = bs

concat as Nil = as

concat (Cons a Nil) bs = Cons a bs

concat as bs = Cons (unsafeHead as) (concat (tail as) bs)


filter :: (a -> Boolean) -> List a -> List a

filter p Nil = Nil

filter p (Cons a as) = if (p a) then (Cons a (filter p as))

      else (filter p as)


-- QuickSort

qsort :: List Number -> List Number

qsort Nil = Nil

qsort (Cons x xs) = (concat

      (concat (qsort (filter (< x) xs))(Cons x Nil))

      (qsort (filter (>= x) xs)))
```

In the provided code in Figure 2, we see a more non-trivial example of Zenith, where we use a more complete set of its features to build a program. As a brief introduction to recursion in Zenith, we reference the Fibonacci sequence algorithm. With recursion, in both Zenith and many functional languages, much the language design focuses on pattern matching, where we can elaborate the base and special cases of recursion through various patterns, as we can see in the fib function.

More notably however is the implementation of the quick sort algorithm, using various list operations. To begin, the Zenith standard library packages the $List$ data type, which itself represents either a pair, as $Cons$, or the empty list as Nil. Lists are then formed as conjoined sequences of these pairs, essentially forming a linked list data structure. As such, we can implement the $concat$ function in the recursive manner shown. Filter is implemented in a similar method, recursively applying a higher order function to the list values, in order to evaluate the condition. Finally, we compose these functions together to form the recursive quick sort function, where we filter lower and higher values of a list and concatenate their sorted result together. Also, as a note on the use of $unsafeHead$ in this section, normally, the head function will return a $Maybe$ type. For brevity however, this figure utilizes $unsafeHead$ for direct access to the List head.

To some degree, these example programs are still relatively simple, and are somewhat constrained by the current implementation of Zenith, and its limited standard library. However, we can also see how we are able to use Zenith to compose functions and programs of increasing complexity. From the implementation of List, Maybe, and other data types, along with Zenith's basic functionality, we are provided a foundational basis for the well-typed lambda calculus, and as such we can develop reasonably similar programs. In the next section, we will evaluate Zenith's current implementation, against its goals of reducing runtime errors.

5.1 Evaluation against Errors

So far in this chapter, we have seen how Zenith is a rudimentary but type safe and functional language. However, based on this usage, it is prudent to evaluate the extent to which Zenith accomplishes its goal of eliminating runtime errors. To begin, it is necessary to elaborate the domain of runtime errors that we evaluate upon. Different languages and systems abstract runtime errors in unique ways, and as such may have a variety of runtime exception categories. Since Zenith is specifically focusing on the Lua programming language, we will first need to identify the categories of Lua runtime errors.

However, the first issue we encounter is that Lua does not provide a centralized document or structure of the different exceptions in the language, such as the exception hierarchy class in the Python documentation (Python Software Foundation). Being frequently included in embedded contexts, Lua does not necessarily operate its own well-defined exception handling system. Rather, when an error is raised, Lua returns control to the host program, which can then react to the error. In the case of the Lua standalone interpreter, this may be a response such as printing out the error, a stack traceback, or some call to a C function. Because of this ambiguity, we will instead roughly follow the exception hierarchy that Python provides as a rough outline of the classes of exceptions that a dynamically typed language may encounter, and then adjust for Lua's semantics when necessary.

Following the hierarchy, exceptions are divided into the various categories they might appear as, such as arithmetic errors and import errors. In this paper, we will group these categories by how Zenith may secure these errors, or by how Lua already eliminates them. First, we can safely ignore consideration for errors that are not present during runtime, and for errors that are not possible in static languages.

Table 1

Runtime Errors in Zenith and Lua

Error type	Example	Lua	Zenith
Arithmetic Error	`1/0`	inf	$Maybe$ type
Attribute Error	`local x = 1` `x.insert(2)`	Error	Type checked
Assertion Error	`assert(n==3,‘msg’)`	Error	N/A
Import Error	`require(‘invalid’)`	Error	N/A (IO)
Lookup Error	`local x = {1,2,3}` `print(x[10])`	nil	$Maybe$ type
Memory Error		Stack overflow	Stack overflow
Name Error	`10 + x`	Error	Type checked
OS Error	`os.remove(‘invalid’)`	Error	N/A (IO)
Recursion Error		Stack overflow	Stack overflow

As we see, Zenith is mostly capable of securing the runtime errors present in Lua, particularly regarding types and data, and the means of accessing data. Furthermore, Lua may return unsafe or unexpected values like nil in specific cases. In these cases, we see how Zenith may abstract this behavior more safely, into a functional data type. However, we can also see that Zenith does not provide much protection regarding IO, which we may expect from the design of Zenith relying on Lua for IO. In regard to Zenith's own safety, we see that the single remaining class of runtime errors are memory related, being memory errors and recursion errors. In earlier chapters, we made the decision to provide a Turing-complete language over an incomplete one, so therefore the presence of recursion and out of memory errors is the natural tradeoff.

However, Zenith does introduce its own unique form of error, where a developer may provide incorrect type annotations for an external Lua function. At the moment, Zenith has no capability for verifying these external annotations. An incorrect annotation may be considered as a logical error but may indeed throw a runtime error depending on the incorrect type. Additionally, Zenith contains a small number of bugs in its current implementation. Most notably, pattern matching is not checked for exhaustiveness. Therefore, it is possible to match on a non-existent case, where in the absence of a default case, will cause a runtime error. However, while this issue is significant and will not be detected by the compiler, it can be mitigated by including a default case where pattern matches appear. Furthermore, as mentioned in Chapter 4, the renderer for Zenith to Lua is relatively basic. An untested issue that may arise is the possibility of namespace collisions between variables, after the compiling process, which may lead to runtime errors. Despite these issues however, based on their infrequency, and from the example programs and errors that Zenith secures, we evaluate that Zenith is a reasonable approach to providing runtime safety for Lua.

CHAPTER 5

CONCLUSIONS

Zenith provides a functional, concise, and most importantly safe means of writing Lua code that may be otherwise unsafe, by using the ideas and work grounded in type theory, functional programming, and language design. By making guided decisions while following these concepts and understanding when to make tradeoffs in the case of Turing-completeness, this paper introduces a programming language uniquely capable of providing safety to the Lua language. While programming in Zenith, writing code and modeling safe systems are all one and the same, and the Zenith programmer can be assured that if the program compiles, it will indeed not error based on the errors we evaluated.

However, there is still a significant amount of future work that may be conducted regarding Zenith. For example, dependent typing, as mentioned, was a notable consideration for the language that did not make it to implementation but would further allow for safety constraints on a program, using its type system. Dependent typing was largely avoided due to the complexity it tends to add to languages, but novel approaches may be possible when considering Zenith's design as a "second language". Furthermore, this paper was largely theoretical and abstract, laying out the groundwork for Zenith to be built upon. However, a programming language is much more than just its design. More work can be investigated into language tooling for Zenith, and greater optimizations can be added into the core compiler and type checker itself. Additionally, a key component that was left out of early consideration for Zenith was usage of the LuaRocks package manager. Languages like Elm exclude much of the community package manager system for their base language, but other languages like TypeScript add support for the common package managers through community efforts. This is a notable decision for Zenith to make in the future, and work can be done on justify whichever choice to make.

Regardless, what this paper hopes to show through this project and review is how we can utilize functional programming and static typing principals to secure a language's runtime and handle the challenges of unsafe code. In addition, the Zenith language is an exploration of secondary languages, and how programming systems can benefit from a separation of responsibilities. In Zenith, these ideas have filled the critical role of safety in a programming system, and as such we hope that Zenith is able to be of benefit to the Lua development space.

REFERENCES

A delightful language for reliable webapps. (n.d.). Retrieved April 1, 2023, from https://elm-lang.org/

Brady, E. (2013). Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming, 23*(5), 552-593. doi:10.1017/S095679681300018X

Czaplicki, E. (2012). Elm : Concurrent FRP for Functional GUIs.

Damas, L., & Milner, R. (1982). Principal Type-Schemes for Functional Programs. *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 207–212. Presented at the Albuquerque, New Mexico.

Dhall configuration language. (n.d.). Retrieved April 1, 2023, from https://dhall-lang.org/

Jones, M. P. (1995). Functional Programming with Overloading and Higher-Order Polymorphism. *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, 97–136. Berlin, Heidelberg: Springer-Verlag.

Jones, S. P. (ed.) (2002). Haskell 98 Language and Libraries: The Revised Report. http://haskell.org/

Jones, S. P. (2007, June). A History of Haskell: being lazy with class. *The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*.

Kozen, D.C. (1977). Rice's Theorem. In: Automata and Computability. *Undergraduate Texts in Computer Science*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-85706-5_42

Payr, L. (2021). Refinement types for Elm. *Linz : Research Institute for Symbolic Computation (RISC)*.

Pickard, Gabriel. (2020, November). Programming languages shouldn't and needn't be Turing complete.

Python Software Foundation. (n.d.). *Built-in exceptions*. Python documentation. Retrieved April 19, 2023, from https://docs.python.org/3/library/exceptions.html

Rice, H.G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society, 74*, 358-366.

Wadler, P. (1995). Monads for functional programming. NATO ASI PDC.