Charles R. Lipford

905502749

**Independent Study Proposal**

Hydra: A lexically scoped, dynamically typed, concurrent, imperative programming language.

**Motivation:**

Since the 1970’s , hardware has advanced exponentially with regards to size and power. It is commonplace for a laptop to have between 4-8 independent cores and it won’t be long before mobile devices have multiple cores as well. Even with all the hardware innovations, however, only a few programming languages have been designed to take advantage of the multiple cores of today’s technology. Go, Erlang, and Scala are all well known for their scaling ability due to their concurrency models. All three of these languages, however, are statically typed. This is where Hydra comes in.

While there are a few statically typed programming languages that excel at concurrency and by virtue parallelism, there are no mainstream dynamically typed languages with true built in concurrency that can be scaled to multiple cores. Elixir is a new language built on the Erlang VM that has very good concurrent support but it is still on the rise and a functional language at its core. Javascript is single threaded, and while different Python and Ruby implementations have both real OS threads and pseudo-threads, they are limited by their global interpreter. The Global Interpreter Lock(GIL) is the problem that comes about when a program in an interpreted language spins off multiple threads but needs access to a global interpreter to be able to run each one. This causes whichever threads do not have the interpreter to block until they acquire the GIL.

Ruby, Python, and Javascript are all used extensively in production as servers, test suites, and the backend behind lots of CLI tools. While many teams choose them for their readability and writeability, the speed and memory performance drop they suffer in comparison to statically typed languages such as C, C++, Java, Go, etc. dissuade many from using them for anything performance critical.

Multi-processing, spinning up multiple processes to do work with data passed around in pipes , is sometimes used to get around this problem but it is very memory and resource inefficient.

With Hydra, the goal is to create a flexible, readable, writeable, and modular language with intuitive syntax and semantics that allows programmers to easily reason about and utilize concurrency primitives to create scalable programs. With that in mind, any and all Hydra interpreter implementations must not fall into the same GIL problem as other dynamic languages. This document outlines a proposal for a language specification and interpreter design that would be memory and speed efficient while allowing for true OS level concurrency and parallelism.

Along with the performance goals of Hydra, syntax, readability, write ability, flexibility, and modularity are at the forefront of its design considerations. In this regard, the programmer should be thought of as the user. Hydra should be designed to let programmers bend and extend it to work as they see fit. Its syntax and semantics will pull from Javascript, Ruby, Go, Julia, Python, Elixir, and a few other languages but will try to avoid their pitfalls as well. While Ruby has a great readable syntax, it also has so many language constructs that it is hard to navigate all of the possible ways to write the same code. Beyond this, both Ruby and Python have “magic methods” that do a lot of work in the background but are a syntactic eye sore and are not as transparent as they should be. Python syntax is also reliant on white space, which can be very hard to keep consistent especially across different editors and development environments. Javascript, on the other hand, is extremely terse but it has scoping problems along with having a single threaded run time. Hydra will take the best syntactic and semantic parts of these languages to construct a clean, intuitive syntax that allows programmers to work quickly and efficiently.

As far as flexibility and modularity goes, it will include a hygienic macro system, pattern matching, list comprehension, tuples, dynamic typing, classes with public and private data, a module system, and user definable language constructs. Its built in types will include Exceptions, Arrays, Enums, Functions(first class, anonymous, closures, and generators), Maps, Strings, Booleans, Integers/Floats, Regular Expressions, and Channels. To promote good object oriented practices, everything, including most primitives will be extensible objects. That way programmers can easily extend built in types to have additional functionality such as keeping values sorted for binary searches or extending strings to have some special attributes.

Compositional inheritance will allow for a class to pull its functionality from multiple super classes. In that way, classes will be thought of in terms of what task they need to perform instead of how to fit them into some hierarchy. It will also promote smaller, more modular code that can be reused to provide functionality when needed. Because the composition is at the language level, it will circumvent the notorious issues caused by mixins in third party libraries causing problems in user code.

**Solution Overview:**

The main issue that needs to be considered when designing an interpreter for Hydra is an implementation that does not use a global interpreter lock. This is necessary to guarantee that the concurrency built into the language can scale to multiple cores without the bottle neck of only one thread being able to be run by the interpreter at a time. This can be accomplished by letting both the main thread and all spun off threads run the same interpreter code. This, in turn, implies that the interpreter code must be reentrant and use no global variables. While these requirements up the complexity of the interpreter, they also give a few benefits. Because all of the threads are pointing to the same code, it only needs to be loaded into memory once. This also means that when a program spawns a new head(starts a concurrent asynchronous function), that the thread it is running on will be able to treat that code as if it were in the main thread. A head could start other heads and so forth.

To allow for the scheduling of heads, communication between different heads through channels, dealing with blocking system calls, and the garbage collector, a compiled runtime must sit underneath the executing interpreter. This runtime would be called whenever a new head is spawned, a new channel is instantiated, read from, or written to, a system call is made, or the garbage collector is run.

The concurrency in Hydra will be handled by an M(os thread):N(head) model. When a new head is spawned by user code, a new thread will be started up to a configurable limit that defaults to the number of physical cores. The use of M os threads allows the os to schedule them on different cores so that concurrency can be parallelized. If the number of threads is already at its max, the new head will be placed in a queue of runnable heads waiting for an os thread to run on. When a head finishes running, all of its associated memory is marked for collection and its underlying os thread is handed off to the next head in the queue. If a head makes a system call, the runtime will try to create another os thread to take a head out of the runnable queue so that the blocking call wont make waiting heads wait for it to finish. However, if the thread count is already at the limit, the head will still be run on the current thread. Channels will be the mechanisms through which different heads communicate. You can think of them as a F.I.F.O. shared queue. They will work similar to os pipes except for some semantics of buffered vs. un-buffered channels. Buffered channels are instantiated with a limit on how many items they may hold. When a head tries to send on a buffered channel that is not full, the send is non-blocking. If the channel is full the call blocks until a receiver pulls an item out. An un-buffered channel can be thought of as a buffered channel that only holds one element. Sending is non-blocking if and only if the channel is empty. For both types of channels, reading from it blocks only when it is empty. Channel communication is still handled by the runtime so that its blocking calls can be handled the same way blocking system calls are. Because channels are shared between different heads, their instantiation is also handled by the runtime so that they can be properly garbage collected.

Now that concurrency has been addressed, flexibility is up next. Most of Hydra’s flexibility will come from a Julia and Elixir inspired hygienic macro system and the semantics that surround functions. The macro system will allow for expressions to be quoted and unquoted so that literal and evaluated forms can be used in a macro. On top of that, Hydra will allow multi-macros that allow for a macro to match multiple smaller macros and their parameters so that they can be combined in a new way. This flexibility will allow for users to define macros that look and feel like language constructs. Functions have two important parts to them. The first is that closures can capture variables both by reference and value. Any variable passed into a closure as a parameter is a reference to that object. Those values are bound to the closure so that any later call to it cannot change their values by passing something else in. If a variable from an outer scope is simply used without explicitly passing it in then its value is copied at the time that the closure is created. This semantic also dictates that you cannot create and call a closure simply by ending it with open and closed parenthesis. You must first put the closure in a variable and then call the variable with the parenthesis. Secondly, there are two representations for closures. The first is taken from Ruby and is called a block. It starts with the “do” keyword and ends with the “end” keyword but does not take any arguments. The second is a simple function. It starts with an open and closed parenthesis with parameters in between followed by a curly brace delimited series of statements. This is used when a closure needs to receive variables as parameters. These two forms of closures work together to both let macros feel like language constructs and give a clean syntax for passing functions as parameters.

**Solution Issues:**

The given high level design has a few holes in it. While it solves the problem of different threads needing a single interpreter instance to run the code, it introduces a new problem. If two different heads are running and call a function on an object of type X, they will have to do a look up in a symbol table. But where should that symbol table be? If we implement head local tables then any changes to the values in a table at runtime would not be reflected in all the heads. This would also cause lots of copying of information. If we use a global symbol table however, we would have to protect it with locks whenever a head had to get information from it and we would be back to square one. Also, garbage collection has to be able to run across all the heads and know which head is using what variable. The last issue is the semantics behind macros and if they are defined on a specific class or module. If they are defined on classes, how do you differentiate between them at runtime efficiently?

**Addressing Solution Issues:**

A simple but efficient solution to the symbol table problem is to have one global symbol table that is constructed once at the beginning of a program. That means that all module definitions and imports, class definitions, high level functions, and macros would have to be parsed and put into the global symbol table before any code starts to be interpreted. This makes the symbol table read only and the need to lock it goes away.

For garbage collection, the solution is not nearly as simple. It consists of two parts. Head local reference counting and an independent GC head that goes around attaching itself to individual heads and garbage collecting them. Each head will contain a list of objects used within the head. When an object goes out of scope its reference count is set to zero but it is still in the list. Then, when the GC head attaches to the running head, it will lock the list so that the running head cannot make changes to it while the GC is collecting free-able memory. Then when it is finished it will unlock the running head’s reference list and move onto the next head. This GC head will also run on the “main” head. The runtime will provide the policy for which running head the GC will attach to next. When a Head exits completely, all of its resources that are not shared with other heads will be collected. If a variable is shared with other heads, updating its reference will require locks.

As far as defining class based vs. global macros, there are many more benefits to going with only global macros. Macros should be as flexible as possible. That being said, they can call a specific function on their operands to guarantee that the operand must have that API satisfied. In this way, many classes could be modified to be able to be used with the macro simply by defining a macro specific function. This would also enforce runtime checking that the macro operand is of the correct type without having to build anything else into the language.