Implementing Novice Friendly Error Messages in Clojure

Charlot Shaw
Computer Science Discipline
University of Minnesota Morris
Morris, MN 56267
shawx538@morris.umn.edu

March 16, 2018

Abstract

Clojures promise as an educational language taught to first time programmers is limited by its error messages, which show its Java underpinnings. The messages are generated by the Java code executing the Clojure program, and thus are focused on the state of the Java implementation of Clojure, which hides the details about the issue in the users code. By using these messages, along with the information obtained via Clojure.Spec contracts, error messages can be reformatted into a beginner friendly, near plain-language representation. By integrating the re-formatting code into Clojures nREPL middleware system, the transformation can be accomplished without disturbing programmers' workflow, and be easily incorporated into Integrated Development Environments. We present a proof of concept implementation integrating beginner friendly error messages into Clojure via nREPL.

1 Introduction

Clojure [2], amongst other functional languages, has gained attention in recent years in part for its management of state and concurrency. Running on the Java Virtual Machine (JVM), Clojure has a number of advantages for beginning students. Its Lisp style syntax and elegant core library work together to let beginners learn easily, while its ties to the JVM keep it relevant as students progress into later courses. However, from an educational perspective, it has a significant flaw in the form of error messages. As Clojure code runs in the JVM, its errors take the structure and terminology of Java error messages, and so are confusing to new students. They can understand the source of the error, but not how the system presents it to them. For example, a user who accidentally called addition on a boolean needs to understand the Java object hierarchy, casting, and the classes involved to fully understand what is meant by "java.lang.classCastException java.lang.Boolean cannot be cast to java.lang.Number", whereas the mental overhead required to understand "In function + the first argument `true` must be a number, but is a boolean" is significantly lower, with the latter being closer to a plain language description of the issue. In order to overcome these problems, we have explored possibilities for integrating customized error messages with common tools in the Clojure programming community.

2 Error Messages in Clojure

Clojure is hosted and interpreted in the JVM, as a Java program. Clojure code can either be loaded by a running Clojure process, or compiled Ahead Of Time (AOT compilation) into Java bytecode. In either case, even simple Clojure code mid-execution is in actuality a complex Java program. When that program encounters any error, including syntax errors, it is thrown as a Java exception. From the viewpoint of the JVM, the entirety of the Clojure process is a part of the users program, and so error messages include large amounts of data about the underlying state of the Java classes that implement Clojure. This surplus information can be useful in debugging, but whole message is still phrased from a Java perspective, requiring familiarity with Java to understand what is happening in Clojure. This undue onus on the beginner, unfamiliar with programming in general, is an unreasonable burden. However, the information present can be leveraged by us to make our improved error messages, though there is not enough information in them to rely on them entirely. A second source of information is preferred when possible, and can be found within Clojure itself, in the form of Clojure Spec contracts.

3 Clojure Spec

The second piece of the information required can be sourced from Clojure Spec [3] which is a recently added library within Clojure, dealing with runtime validation of the structure of data within a program. These take the form of specifications, or contracts and can be

applied to any data structure, including the arguments supplied to a function. For example, a function dealing with a list have have a Spec contract enforcing that all arguments used as list indices are non-negative integers. Importantly, these contracts provide detailed information about how the actual values differ from those required. By providing Spec specifications for the core library of Clojure, we can cover every function that a novice is able to call, and thus catch any miscalled functions with detailed Spec error messages. This information can be captured by our system, and as a result lets us access information such as the actual values that caused the exception, a detail often omitted by default Clojure errors. Note that Spec only provides errors for function arguments; for syntax errors or other kinds of runtime exceptions, we need to source our data from the original Java error messages. Between Spec error messages and the original Java error messages, we have enough information to leverage the tools built by the University of Minnesota Morris research group [4]. However, up until now it was not clear how this system could be integrated with the larger Clojure ecosystem.

4 Clojure, REPL and the IDE

Clojure is homoiconic, meaning that a Clojure program is also a Clojure data structure. Evaluating every value in that structure is equivalent to running the program. This allows a Clojure process to read in data, process it as a program, display the results back to the programmer, and keep running, awaiting the next input. This common sequence of steps is referred to as the Read-Eval-Print-Loop (REPL). Programmers use it to interactively test and build up individual parts of their programs. This pattern of development is referred to as REPL-Driven and is commonly used in Clojure community. Most Clojure Integrated Development Environments (IDEs) feature REPLs for user convenience. Outside of developer experimentation, the REPL mechanism is also used by the Clojure process to load in new Clojure code for execution in a running program, omitting the Print step if necessary. Spanning both evaluation of code, and the preparation of output back to the user, the REPL is a prime location for our system to be implemented. However, building our own custom REPL is not advisable, as it limits the integration possibilities, needing IDEs to be customized to support it. Instead, the ideal position for our code is within a module for a popular, expandable REPL implementation.

5 nREPL

nREPL [1] is a community standard implementation of the REPL concept, using a Client/Server model. It is integrated into Clojure project management software, as well as most commonly used IDEs that support Clojure. nREPL works by passing messages from the client, to be executed on the server, with the results being returned to the client for display. It allows a set of middleware to modify messages, providing utilities like interruptible evaluations or independent user sessions. Custom middleware, such as ours,

can be added by simply modifying a projects configuration file. Once our code is added, it listens for messages coming back from the server marked to be displayed to the users as errors. It customizes the error messages, and then reinserts them into the middleware stack to reach the user. As it is using the same message propagation scheme as an unmodified nREPL, IDEs should pick up and display the new error messages with little trouble. This also does not change or affect anything on the server side, so there are no side effects.

6 Conclusion

Between Clojure default error messages, and Clojure Spec we have all the information needed to modify error messages, using the tools created by prior works of the University of Minnesota Morris research group. Using nREPL middleware lets us place our code seamlessly into the beginners workflow, in a way that would not violate the expectations of more advanced users. Taken together our system shows a proof of concept for how Clojure error messages can be modified for beginners via nREPL, a necessary step in making Clojure an educational language for first time programmers.

7 Acknowledgments

The author thanks the project adviser Elena Machkasova. The work was supported in part by Morris Academic Partnership (MAP) stipend at UMM.

References

[1] EMERICK, C., AND CONTRIBUTORS. nREPL. `https://github.com/clojure/tools.nrepl`. Accessed: 3/16/18.

[2] HICKEY, R. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages* (New York, NY, USA, 2008), DLS '08, ACM, pp. 1:1–1:1.

[3] HICKEY, R. clojure.spec - Rationale and Overview. `https://clojure.org/about/spec`, 2016. Accessed: 3/16/18.

[4] SONG, M., AND MACHKASOVA, E. Improving clojure error messages for programming novices with clojure.spec. In *Midwest Instruction and Computing Symposium* (2017).