Improving Clojure Error Messages for Programming Novices with clojure.spec

Myeongjae Song and Elena Machkasova
Computer Science Discipline
University of Minnesota Morris
Morris, MN 56267
songx823@morris.umn.edu, elenam@morris.umn.edu

Abstract

Functional programming paradigms have been getting mainstream attention recently because of their elegant concurrency handling and conciseness of source code. University of Minnesota, Morris has a strong tradition of using functional programming languages, in particular those in the Lisp family, in the introductory CS curriculum. A Lisp language is often the very first language that our majors encounter. Clojure is a relatively new language in the Lisp family that is becoming increasingly popular nationally and worldwide. While it would be desirable to use Clojure as a beginner language in CS curriculum, there are several challenges in adopting it for novice programmers. One of the most significant challenges is its error message system which uses terminology and concepts that are unfamiliar to beginners. A current project at UMM aims at developing a system of beginner-friendly error messages for Clojure. A recently added system of contracts in Clojure, known as clojure.spec, is the evidence that the Clojure community at large is also interested in improving error reporting. This system came out in summer 2016, and we have been exploring its ability to facilitate more robust and flexible error messages, particularly ones aimed at helping beginners. This paper describes our current system of providing beginner-friendly information in error messages and presents our recent work on switching to clojure.spec for this purpose.

1 Introduction

Clojure is a relatively new programming language in the Lisp family, introduced by a developer Rich Hickey in 2007 [5]. It is a Lisp language with a dynamic type system. It compiles to Java bytecodes and runs on the JVM. This means that a Clojure program can access any Java libraries. It quickly gained a worldwide popularity in industry and has a large and very active community of developers. As a Lisp language, Clojure follows functional methodology with an emphasis on immutability, functional abstraction, and recursion – concepts that are essential in CS education, as discussed in [3]. As such, Clojure could be suitable as a first language taught to computer science students, continuing a rich tradition of MIT Scheme [1], on par with the Racket programming language, also in the Lisp category [2].

However, there are several issues that make introducing Clojure to beginner programmers quite challenging. The most notorious of these issues is the lack of beginner-friendly error messages. Our research group at University of Minnesota, Morris (UMM) has been working on developing a system of error messages that are helpful to beginners. As our work on this system was progressing, Clojure has introduced a new feature that would allow us to interface with Clojure core much easier. This feature is a system of contracts for data validation, known as clojure.spec. Thus our current goal is to transition our older ad-hoc system of collecting information about an error to clojure.spec. This paper presents a comparison between our two approaches and demonstrates our current progress in this transition, including some challenges that we have overcome.

The paper proceeds as following: Section 2 introduces Clojure and clojure.spec, Section 3 presents our system of error processing before clojure.spec, Section 4 discusses the new approach, and Section 5 summarizes our current progress and discusses future work.

2 Background

This section is an introduction to Clojure and the clojure.spec library.

2.1 Clojure

Clojure is a Lisp-dialect with some distinct features in comparison to other popular languages, such as Java or Python. One such feature is the use of prefix notation in Clojure syntax. In Clojure, every instruction or expression is enclosed in parentheses. To invoke a function, you need to specify a function name right after the opening parenthesis and then list all arguments before the closing parenthesis. Parentheses are even required for functions that do not take any arguments. A simple addition of 3 and 4 in Clojure looks like:

```
(+ 3 4)
```

In Clojure, + is a just function, not a special operator, and the same is true of what would be other common operators in other languages. This syntax might look odd, but it is very efficient and easy to parse for machines since the system does not need to calculate operator precedence. Moreover, it easily accommodates a variable number of arguments, which means you can provide any number of arguments to some functions, such as the + function. For instance, both `(+ 3 4 5 6)` or `(+ 3 4 5 6 10 15)` work.

Defining a function is very intuitive in Clojure. Inside a pair of parentheses, you list `defn`, the function name, the argument list, and the expression. An example of a function that multiplies a given number by two looks like:

```
(defn multiply-by-two [n] (* n 2))
```

In this example:

- `defn` indicates the declaration of a function.
- `multiply-by-two` is the function name.
- `[n]` is the argument list: there is only one argument named `n`.
- `(* n 2)` is the expression that multiplies n by 2. The result of this expression will be returned from the function.

Calling this function with an argument 3 would look like this: `(multiply-by-two 3)`, and returns 6.

Another functional programming feature that Clojure adopted is immutable variables. In Clojure, once you bind a name to a value, you cannot change what the name is bound to. This is the main source of confusion for developers from imperative programming backgrounds since they are used to changing variables to store different values. The main benefit of using immutable variables is minimizing side effects, which means developers do not need to worry about variables being modified from unanticipated operations. There are mutable variables in Clojure, but they need to be explicitly declared and modified with special syntax. Thus, side effects are not as serious concern as in imperative programming languages.

Among various data structures that Clojure supports, sequences are unique in the way they behave. In particular, a sequence can be lazy. In a lazy sequence, element(s) do not exist when the sequence is first created, but they can potentially be computed later when they are needed. In other words, lazy sequences are like a set of rules capable of generating as many elements as are needed. Such sequences just exist as a lazy sequence type until some functions force them to be evaluated. Because of its laziness, it is possible for a lazy sequence to represent an infinite sequence. The following is a function that returns a lazy sequence representing the Fibonacci sequence. In this case, the starting values are `n1` and `n2`. Theoretically, it could generate elements of the Fibonacci sequence indefinitely.

```
(defn fibonacci [n1 n2]
   (lazy-seq (cons n1 (fibonacci n2 (+ n1 n2)))))
```

Another commonly used Clojure data structure is hash maps. It is essentially a collection of key - value pairs. It is very useful when representing data with relationships. Following is an example of a hash map to represent a college course: course number is paired with a course name. The key and value are separated by a space, and each key - value pair is separated by a comma.

```
{:csci3501 "Algorithms", :csci2101 "Data Structures"}
```

In this example:

- The curly braces represent that this is a hash map.
- `:csci3501` and `:csci2101` are the keys.
- `"Algorithms"` and `"Data Structures"` are the values.

2.2 Introduction to clojure.spec

The cloure.spec library introduced in Clojure 1.9 in May 2016 [4], and technically is still in development. It is a contract system that allows Clojure developers to specify and validate expected data at runtime. In addition to data validation, clojure.spec provides easy-to-parse error messages when the data is invalid. To start using specs, we need to define a spec first. Each spec is just a specification for data. You can use `s/def` to define specs.

The following are some simple spec definitions with Clojure predicates:

```
(s/def ::check-int integer?)
(s/def ::check-function ifn?)
```

You can use the above spec definitions to check data types with `s/valid?`, which returns a boolean. If the data satisfies the spec, `s/valid?` returns true. Otherwise, it returns false. For example, `(s/valid? ::check-int 3.5)` will return `false` because `3.5` is not an integer number.

It is possible to assemble simple specs to create a more complicated spec. For instance, we can write a spec that verifies the argument types of a function by combining `::check-int` and `::check-function` with `s/cat` which stands for concatenation.

```
(s/def ::check-int-function
    (s/cat :first-arg ::check-int
           :second-arg ::check-function))
```

If the above spec is attached to a function `my-function` that takes two arguments, `(my-function 4 +)` will be valid because the first argument `4` is an integer and the second argument + is a function.

One really powerful feature of spec is the use of regular operators such as `s/cat`, `s/*`, and `s/?`. As we have mentioned above, `s/cat` stands for concatenation. `s/*` represents zero or more repetitions of a data pattern, and `s/?` represents zero or one of a data pattern.

Using these regular operators, we can describe a variety of data structures, including lazy sequences and nested data.

To define input and output specifications for functions, clojure.spec uses `s/fdef`. Even though it is possible to define the output spec for a function, it is only usable in spec testing. For our purposes, we will be focused on the input verification. To start spec input validation of a function, it should be turned on by `(stest/instrument 'function-name)`. Otherwise, the spec is completely ignored at runtime. Let's take a look at an example of `s/fdef`. The following is a simplified definition of `even?` function and a possible function spec for it. Whenever `even?` function is called, the spec is invoked and checks if the argument is an integer.

```
(defn even? [n] (= (mod n 2) 0))
(s/fdef even?
     :args (s/cat ::check-int integer?))
;; turn on the function spec
(stest/instrument 'even?)
```

The spec will allow the function to be called only when its parameter is an integer. Any other type of the parameter would produce a run-time error.

3 Clojure error messages

3.1 Overview of Clojure error messages and our approaches

Since Clojure compiles to Java bytecodes and runs on the JVM, Clojure error messages are just Java exceptions and uses Java datatypes and terminology. For instance, `(+ 2 true)` is an attempt to add a number and a boolean. Since Java does not automatically convert booleans to numbers, this code results in an error:

```
ClassCastException java.lang.Boolean cannot be cast
to java.lang.Number
clojure.lang.Numbers.add (Numbers.java:128)
```

This is quite cryptic for beginners since they are not familiar with a term "class" for a datatype, "casting" for type conversion, and "exception" for an error. Moreover, even if a term "boolean" has been introduced to them for true/false values, the prefix `java.lang` does not correspond to their experience. Also, note that the class `Number` has a prefix `clojure.lang` which is different from `java.lang` . To worsen the confusion, instead of referring to the function +, the error message refers to the method `add`, and the class in which it is defined is `clojure.lang.Numbers`. Those familiar with Java may identify this method as a static method (mostly based on the naming convention: `Numbers` is pluralized). For new Clojure programmers not familiar with Java, however, the message carries very little, if any, information. Even if these new programmers memorize common

error messages terminology after a while, it would still not have any grounding in their experience.

Clearly, this situation is not acceptable in an introductory computer science class. Thus, one of the directions of our project is to provide a set of error messages that would be more consistent and meaningful in the context of novice programmers experience.

Our error messages translate standard Clojure error messages into terms that are less confusing for beginners. For example, the above-mentioned expression `(+ 2 true)` would result in the error message

```
In function "+" the second argument "true" must be a number,
but is a boolean.
```

In this error message it is clear what function is being called, which argument is causing a problem, what is expected, and what is given.

Our approach to replacing standard error messages has two main cases and a default case:

1. For commonly used functions we provide a direct assertion to check if we are passing the right number and types of parameters, and report an error if we do not. These errors are more informative than default errors.
2. For other cases we perform a lookup of an error message in our "dictionary" using pattern-matching, and replace the wording so that it is more understandable to beginners.
3. For rare cases that are not listed in the dictionary we have no choice but to report an error as it is.

For this paper we are focusing on the first case: providing an assertion to check the number and type of the arguments. We discuss how we have been handling this situation in the past and how using clojure.spec allows us to collect the same information as before (and sometimes more) in a way that is less error-prone.

3.2 Assertions for common Clojure functions

The approach to providing assertions to check function arguments that we used prior to incorporating clojure.spec was to write our own function with the same name as a standard one, provide a precondition for it to check if the arguments are valid, and if they are, call the function provided by core Clojure. Here is an example of this approach used for function + that specifies that it can take any number of arguments, but all of these arguments must be numbers:

```
(defn + [& args]
   {:pre [(check-if-numbers? "+" args 1)]}
   (apply clojure.core/+ args))
```

In this example:

- the name of the function is +,
- the `[& args]` is the arguments list. The `&` sign indicates that what follows is a list of arguments of arbitrary length.
- The next line (with `:pre` in curly braces) is the precondition, followed by a list (in square brackets) of conditions to check.
- In this case there is only one condition to check. It is given by our own function `check-if-numbers?` and passing to it `args` (the arguments passed to +), the name of the function we are checking, which is "+", and the starting number of the arguments being checked (in this case we are checking all of the arguments, so the starting number is `1`).
- If `check-if-numbers?` returns true, the body of the overwritten function + gets executed. If it return false, an assertion error is thrown by the precondition.
- The last line `(apply clojure.core/+ args)` applies the + function in `clojure.core` to the list of arguments. One can think of `clojure.core/+` as a path to the function + in the core package of Clojure.

Clojure preconditions don't record enough information to generate a helpful error message. In particular, they don't record the name of the function they are attached to or the argument number. Thus our custom-made predicates, such as `check-if-numbers?`, record all necessary information about failing arguments to augment the preconditions.

Before spec there was no good way of packaging necessary information into a failed assertion exception itself, so we used a global variable (known as an *atom* in Clojure) to store this information. The stored information included:

- The type that the predicate is checking for, such as a number in the case shown above,
- The actual type of the argument,
- The value of the argument,
- The name of the function for which the assertion fails,
- The number of the argument, such as first, second, etc.

Revisiting the example above, the call `(+ 2 true)` will make the predicate `check-if-numbers?` be called with the function name `"+"`, the arguments list `(2, true)`, and the starting number of an argument: 1. The predicate then checks the first argument in the list, `2`, and it is indeed a number. Then a recursive call is made with the second argument, `true`. The argument number is incremented by 1 to indicate that it is a second argument. Since `true` is not a number, failure information is recorded in the global variable: the function name is +, the "offending value" is `true` and its type is `boolean`, the expected type is `number`, and the argument number is `2`.

After the information is recorded, the predicate returns `false` to indicate that the precondition has failed. Because the precondition failed, an exception `AssertionError` is thrown to the place of the program that has made the call to + with a wrong argument.

The exception is then caught. Based on its type `AssertionError`, it is passed to a handling function that retrieves the information from the global variable and forms the error message:

```
In function "+" the second argument "true" must be a number,
but is a boolean.
```

The global variable is then "cleared", i.e. the information is deleted from it. This is done so that this information is not accidentally retrieved later when it would become irrelevant and confusing.

In addition to `check-if-numbers?` function, we have similar predicates to check if an argument is a function or a sequences or any other type that may be required. Note, however, that there are not as many type requirements in Clojure as in a statically typed language, such as Java, so less than a dozen such predicates are needed.

While this approach produces a correct error message, it has drawbacks: using a global variable to pass this information from one part of the program to another may be problematic in a multithreaded program. This process would also be fragile if multiple errors are taking place in a cascading fashion, i.e. handling one error triggers another. The switch to clojure.spec eliminates potential inconsistencies between the global variable and the error being handled.

3.3 Lazy sequences and challenges in argument printing

As mentioned in Section 2.1, Clojure has lazy sequences. Lazy sequences are values in Clojure, so they can be passed to other functions unevaluated until their elements are needed. This means that if we pass a lazy sequence (for instance, one generated by a function `(constantly 5)` that produces a lazy infinite sequence of 5s) instead of a number, the value will not be evaluated when an error is detected, it will just be kept as a lazy sequence.

However, one of the functions that forces an evaluation of a lazy sequence, is `print`. Thus we have to be careful to not print an infinite (or just very long) sequence accidentally when printing the error message. We handle this issue with a preview function that evaluates only the first up to 10 elements of the sequence.

Switching from this approach to clojue.spec has created some issues that we will discuss in Section 4.2

4 Improving error messages with clojure.spec

4.1 Benefits of using clojure.spec

If there is no spec attached to a function, and the function fails, it throws an error which has the information about the type of error, error message, and stack trace. The value that

caused the error is rarely given in the error message. Let's say a user tried to add `2` to `true`. Because addition only works for numbers, `true` of type boolean is the value that caused the error or the 'failing value' in this case. Even when the failing value is given, we still need to parse the specific value out of the error message string. The same problem occurs when it comes to the line number indicating where it failed. Because an accurate line number is not given in the error message, users need to go through the stack trace to find where the error actually happened. We used to handle these issues by providing assertions as we explained in Section 3.2.

Unlike default Clojure errors, if a spec-attached function fails, an exception object `clojure.lang.ExceptionInfo` is thrown. This exception object has a hash map of failure information, which can easily be parsed with Clojure `ex-data` function. A simplified hash map of a spec error for the `(even? false)` function call might look like:

```
{:clojure.spec/problems [:pred integer?, :val false, :in [0]}],
 :clojure.spec/args (false),
 :clojure.spec.test/caller {:file "spec_error.clj", :line 87}}
```

The stored information included:

- `:clojure.spec/problems` containing the predicate, failing value, and location of the value (the argument number, in this case 0, which indicates that it is the first argument).
- `:clojure.spec/args` containing all the arguments of the function.
- `:clojure.spec.test/callers` containing the file name and line number.

One prerequisite to provide user-friendly error messages is having enough information about the function failure. Since default Clojure error does not provide sufficient data such as the name of failing function or the failing value, our research group had to use a global variable to store this information, as we explained in Section 3.2. With clojure.spec, we no longer need a global variable because the hash map of `clojure.lang.ExeptoinInfo` object has much more information about the failure compared to the default Clojure error. The information that the embedded hash map provides, contains which predicate failed, what the failing value was, and what the number of the argument was (if the function takes multiple arguments). In addition, they can be easily parsed, as we mentioned earlier.

4.2 Approaches to improve error messages

With clojure.spec, we have used two different approaches to improve error messages. One way is using a function spec. If function specs are attached to the Clojure core functions, the spec error is thrown when some core function call fails. Then, we can simply parse necessary information from the hash map embedded in the spec error, and process it before we show them to the user. For instance, `min` function takes one or more numbers, and

returns the smallest number. We can verify the input of the `min` function by providing a function spec like the following:

```
(s/fdef clojure.core/min
    :args (s/cat :check-numbers (s/+ number?)))
```

This function spec definition checks if `clojure.core/min` function takes at least one argument and if all arguments are of type `number`.

The main advantage of using a function spec is that we do not need to overwrite Clojure core functions. We can simply attach our function specs to the Clojure core functions. It is also very flexible in that it can be turned off if necessary. We eventually decided to use another approach using spec assert, after we discovered some issues with the simpler approach of just using a function spec. One serious issue is that function specs do not work for inline functions, which are a type of function that is inserted by the complier when they are used. Many of Clojure core arithmetic functions are inline functions for better performance. Beginners often use arithmetic functions, including +, −, and ∗, so not being able to use specs for them is a huge downside of the function spec. Moreover, function specs have problems dealing with lazy sequences. When a lazy sequence, that has an error inside, is passed to the function as an argument, the Clojure default error is thrown. For example, let's revisit the `even?` function that we explained in Section 2.2. The `even?` function has an attached spec.

```
(even? (map even? [3 4 true]))
```

The function `map` normally takes two arguments, a function and a collection of elements. What it does is simply apply the given function to each of the collection's elements. If there is an expression `(map even? [1 2 3])`, the `even?` function is applied to 1, 2, and 3. As a result of its computation, `map` returns a lazy sequence of false, true, and false because only 2 is an even number.

Let's go back to the above example using two `even?` functions. First, the argument of outer `even?` is checked. Then, the lazy sequence `(map even? [3 4 true])` is evaluated in some internal Clojure spec checking space. The problem is the function spec for `even?` does not exist when the inner `(even? true)` is evaluated. Therefore, the default `java.lang.IllegalArgumentException` is thrown.

Because of the issues mentioned above, our research team is currently using `s/assert` in lieu of the function spec. This approach is very similar to our pre-spec approach that uses `:pre` and requires overwriting Clojure core functions. However, we can solve the issues concerning the inline functions in that the `s/assert` checks the overwritten function's input before the call to the core function is made.

Following is the overwritten `min` with spec assert. The function `do` just evaluates the expressions in order, which means `s/assert` is checked prior to the function call to `clojure.core/min`.

```
(defn min [arg1 & args]
   (do
```

```
(s/assert (s/cat :check-numbers (s/+ number?)) [arg1 args])
(apply clojure.core/min arg1 args)))
```

Just like the function spec, `s/assert` starts validating data once the switch is turned on by `(s/check-asserts true)` Because our `s/assert` is embedded inside the overwritten function, lazy sequence checking is not an issue anymore as in function spec.

5 Conclusions and future work

5.1 Conclusions

Clojure.spec provides a more appropriate approach for generating meaningful error messages than our previous ad-hoc approach. It automatically collects information necessary to provide useful information about causes of errors. It is also significantly more flexible: we can easily control what kind of information we present to programmers. This opens a possibility of creating multiple levels of error messages, depending on programmers' experience.

In the process of working with clojure.spec we have resolved several technical challenges, such as dealing with inlined functions and lazy sequences. There are still a few unresolved issues, in particular related to multiple errors (errors that happen when evaluating a failing value for another error). However, with clojure.spec being still in development we may just need to wait until they are resolved by the Clojure community.

Overall, using clojure.spec has a promise of creating a robust and customizable system of error handling that relies on features embedded in the language itself, rather than in a separate system.

5.2 Future work

There are still many challenges to overcome before Clojure can be easily adopted in an introductory CS classroom. One of the future directions of our work is to continue evaluating what error messages are helpful to beginners. Another significant roadblock is a lack of a beginner-friendly IDE for Clojure, despite the effort of the Clojure community to develop one. Another challenge is to figure out which features of Clojure should be available to beginners: another relatively new Clojure feature, known as transducers, makes certain expressions that beginners can write by mistake be syntactically valid, but almost certainly unintended and confusing. Whether it is possible to "hide" these features from beginners, ideally by using clojure.spec, is a subject of future work.

References

[1] ABELSON, H., AND SUSSMAN, G. J. *Structure and Interpretation of Computer Programs*, 2nd ed. MIT Press, Cambridge, MA, USA, 1996.

[2] FELLEISEN, M., FINDLER, R. B., FLATT, M., AND KRISHNAMURTHI, S. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, Cambridge, MA, USA, 2001.

[3] FELLEISEN, M., FINDLER, R. B., FLATT, M., AND KRISHNAMURTHI, S. The structure and interpretation of the computer science curriculum. *J. Funct. Program. 14*, 4 (July 2004), 365–378.

[4] HICKEY, R. clojure.spec - Rationale and Overview. `https://clojure.org/about/spec`. Accessed: 3/24/17.

[5] 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.