Improving Error Messages in the Clojure Programming Language

Brian Goslinga, Eugene Butler, Elena Machkasova (adviser)
Computer Science Discipline
University of Minnesota Morris
Morris MN, 56267
gosli008@umn.edu, butle250@umn.edu, elenam@umn.edu

Abstract

Clojure is a promising language for all levels of practical and instructional computer science. Its clean implementation of concurrent features and its performance-aware language implementation make it a promising language for software development. Unfortunately, Clojure's often confusing error reporting acts as a major confounding factor to its practical and educational use and may complicate attempts to both teach and meaningfully employ the language. This paper explores how the error messages produced by Clojure can be improved and describes work done towards this goal.

1 Introduction

Error message efficacy is an important, though often overlooked, consideration when tackling the problems of modern software engineering. Error messages that are difficult to understand and therefore difficult to resolve are daunting for experienced and novice programmers alike [4]. The Clojure programming language's error messages represent perfect examples of this phenomenon.
In this paper, we first present a few basic principles for clear, understandable error reporting. Next, we examine Clojure's current error messages and examine the factors that lead to their inability to clearly report relevant information. We then present improved versions of the original error messages, versions that better follow our basic principles. Finally, we present opportunities for future work and deliver our conclusions.

1.1 Related work

We have found that scholarly work on the usability of error messages is noticeably lacking. During our search for work regarding systematic approaches to error message quality evaluation, we only found [4].
In [4], Traver suggests two general approaches to the problem of unhelpful error messages. Programmer-driven approaches to error improvement involve each programmer creating their own sets of error messages and solutions based on past, personal experience. Compiler-driven approaches involve improving the error reporting of the compiler itself, whether these improvements take the form of improved stack traces or next-generation affective computing. While our approach involves the use of programmer feedback, it is focused on improving compiler-generated error reporting.
[1] is also of relevance to our work, providing an example of a language improvement done with error messages quality in mind. Its description of the `syntax-parse` macro definition facility in the Racket programming language outlines a radical, new concept for error reporting. The `syntax-parse` system's error reporting is informed by the idea that messages generated by the system should be based on documented concepts rather than, as is true of conventional error reporting, implementation details. By doing so, [1] is able to provide clearer and more useful error output to the programmer.

2 Background

2.1 Overview of Clojure

Clojure is a relatively young programming language first introduced in 2007. It is a dynamically-typed functional language in the Lisp family and was designed with an emphasis on support for concurrent programming. Clojure provides literal syntax for multiple collection types, including lists, vectors, maps, and sets (see Figure 1). Lists are collections that directly implement Clojure's sequence interface. Vectors are indexed collections one can retrieve items from by providing integer indices. Sets are collections, but only of

Type	Syntax
List	`(1 2 :keyword "string")`
Vector	`[3 7 :foo "bar"]`
Set	`#{:dog :cat}`
Map	`{:cat "cute", :dog "loud"}`

Figure 1: Literal collection syntax in Clojure

```
(defn frequencies
  "Returns a map from distinct items in coll to the number
  of times they appear."
  [coll]
  (reduce (fn [counts x]
            (assoc counts x (inc (get counts x 0))))
          {}
          coll))
```

Figure 2: Example of Clojure source code

unique items. Maps are collections of keys linked (mapped) to values. Keywords, indicated by the `:` prefix, are simple symbolic identifiers that always evaluate to themselves. They are used as unique tokens to be passed around in a program or to label other elements. Clojure makes full use of its richer syntax, resulting in better syntactic delimitation than most other Lisp dialects (see Figure 2). `fn` is the syntax for creating an anonymous function, parentheses delimit the function definition, and square brackets indicate function parameters. `assoc` associates one or more keys in a map to values and `reduce` returns the result of applying an operation to every element in a collection. This code creates a map in which the keys are each unique item in a collection and the values linked to each key are the number of times each item appears within the collection.

All of Clojure's built-in collection types are persistent data structures. This means that they are immutable, so they cannot be modified "in place", but have fast "updating" operations that do not degrade the performance of either the old or the new version. This is done through structural sharing of the majority of the old and new versions of the data structure. Clojure's lists are implemented like Lisp's traditional lists. Clojure's vectors, maps, and sets are built out of 32-way trees, allowing for fast access.

Clojure is built around a sequence abstraction, and all of the built-in collection can be easily turned into a sequence. Clojure also features lazy sequences. This makes the many practical advantages of lazy evaluation available within a strictly evaluated language. Many of Clojure's built-in functions act on sequences (and will coerce their appropriate arguments into sequences) and return lazy sequences.

Clojure handles mutable state by introducing reference types. These types point to an object and can be updated to point to a new object. All update operations on reference types have concurrency semantics, and there are a number of reference types for different concurrency semantics. The `ref` reference type makes updates by using software transactional

```
(import '[javax.swing JFrame JLabel])
(doto (JFrame. "Example")
  (.add (JLabel. "Hello, world!"))
  .pack
  .show)
```

Figure 3: An example of Java interop in Clojure

memory, the `atom` reference type makes updates using compare-and-set, the `var` reference type stores its value using thread-local storage, and the `agent` reference type uses asynchronously queued updates.
Clojure is also designed to be hosted on a platform. The main implementation is hosted on the Java Virtual Machine (JVM). Because the JVM-based Clojure is the "official" version, we shall assume the use of that version in this paper. Clojure uses Java types when appropriate (for instance, strings in Clojure are `java.lang.Strings`). Clojure also provides several interop forms to make working with and using Java code easy (see Figure 3 for an example). Clojure allows the programmer to easily make use of Java Swing classes to create a window and text label populated with the text "Hello World!". Clojure is also designed with performance in mind, and provides several functions for accessing host features such as arrays for fast code. As a result of Clojure's hosted nature and commitment to speed, several host features make themselves visible in the language. Java's lack of tail-call optimization means that Clojure does not provide tail-call optimization. As a workaround, Clojure provides the `loop` and `recur` special forms for optimized self-recursion [3]. As of Clojure 1.3[1], Clojure uses primitive long math with bigint contagion. Clojure performs overflow checks by default and will throw an exception if overflow occurs.
Clojure uses immutable data structures by default. Its clean implementation of concurrent features and its performance-aware language implementation make it a promising language for software development. Despite being a new language, Clojure has generated significant attention in industry, as suggested by anecdotal evidence regarding the rise in questions about Clojure at job interviews. Furthermore, Clojure may also be an attractive alternative to Scheme/Racket for introductory courses due to its richer syntax, and ability to allow for smoother transitions to later Java-based courses.

2.2 Issues with Clojure error messages

As a result of Clojure's hosted nature, Clojure's error messages are presented using the host's underlying mechanism (exceptions), with type errors being presented using the corresponding concept on the host (`ClassCastException`). This reliance on the host is extremely problematic, as it makes Clojure's current error messages less intelligible within the context of one's Clojure instructions. The error messages frequently expose implementation details when these details are not relevant. This tends to cause further confusion about the reported error rather than enlightenment. An example of this form of suboptimal error presentation is found in the stacktrace presented in [2]. The stack trace provides co-

[1]Clojure 1.3 has not been released as of the writing of this paper

pious implementation detail, but is of almost no use in explaining the programmer's error. In addition to the potentially confounding interactions created by the hosted nature of Clojure, Clojure's error messages are flawed in two other important ways. First, Clojure's error messages, in some circumstances, do not yet yield enough information about the problem to aid successful debugging. Second, the error messages occasionally do not respect the abstraction that the user is using and, instead, refer to less relevant implementation details. The current messages for several errors can be found in the table in appendix A.

3 Improving error messages

After examining various error messages and their possible improvements, we discovered that Clojure error messages could be approximately classified into one of three groups: those caused by type casts, those caused by invalid arguments, and those dependent on context.
In section 3.2 we describe the issues with the current error messages caused by type errors, and how we improve these error messages. In section 3.3 we look at the issues presented by the larger class of error messages caused by invalid arguments (this excludes those caused by type errors), and propose ways to improve these error messages. Finally, in section 3.4 we examine the issues and hard problems faced when trying to improve very context-dependent error messages, and present possible solutions and their trade-offs.
For examples and classification of several error messages, see the table in the appendix A. The error messages in the table are referred to by number throughout the course of this paper.

3.1 Basic principles

Our improved error messages are designed around several basic principles. Adherence to sound, basic principles helps to ensure that our improved error messages are consistent and predictable. Our basic principles are designed around a controlled exposure to implementation details, and quality of implementation.
We feel that many of current error messages in Clojure solely display implementation details. This has the effect of making error messages confusing for many newcomers, as understanding these error messages requires that one understands the implementation details of the language. At the same time, hiding all of the implementation details would make it harder for newcomers to learn about the implementation of Clojure, and would cause frustration during the rare occasions that knowing implementation details would be helpful.
Our basic principles address the above concerns. Our improvements to Clojure's error reporting should make the new error messages understandable even to those without a developed understanding of the language's implementation details. At the same time, the new error messages should present the implementation details so that the error messages are useful when details are needed, so new Clojure programmers can gradually learn about the implementation details, if they so desire.

Given our approach to improving the error messages, there are several possible ways to implement the improvements. Each of which comes with various trade-offs. As mentioned in subsection 2.1, Clojure has a strong commitment to performance. Given that errors should be exceptional, it would be very unfortunate to penalize correct code with a runtime cost for better error reporting. We should thus strive to avoid increasing the runtime cost of operations if possible, and increase the costs minimally otherwise. At the same time, we also wish to make minimal changes to the implementation of Clojure. This is based on the assumption that the less code we have to change, the easier it would be for our work to be integrated and used. To satisfy these constraints on our implementation, our primary method of improving Clojure's error reporting is to change how the thrown exception is displayed to the programmer. This means that we only have the original error message and a stack trace to work with, but this also means that we do not need to change the implementation of Clojure (and thus possibly add additional runtime cost) in a majority of cases. We therefore design our improved error messages so that, in the majority of cases, they can be produced using only the original error message and the stack trace.

3.2 Clarifying type cast exceptions

One of the most common error messages in Clojure is the `ClassCastException` error message. These messages are generated by the JVM when the user attempts to cast an object to an incompatible class. Due to the number of them that occur, we distinguish these error messages from invalid argument exceptions. See appendix A for a listing of error messages of this type.
A fairly typical example of such an error message is error message 1 (see Appendix A). In this error message, the troublesome code is

```
(+ {:a 1, :b 2} 3)
```

The current error message that this code snippet results in is

```
ClassCastException clojure.lang.PersistentArrayMap cannot be
                  cast to java.lang.Number
```

There are several issues with this error message. First, the "cannot be cast to" language in the error message does not make sense in the context of Clojure as the programmer does not make explicit casts. Second, although the actual type names are useful to know, this error message does not give much indication as to what entities these type names correspond to.
To address the first issue, "Cannot use ... as ..." language would be more appropriate for Clojure. Switching the previous error message to this form yields:

```
             ClassCastException Cannot use
clojure.lang.PersistentArrayMap as a java.lang.Number
```

To address the second issue, we add an informal, intuitive description of the type name that we term *fuzzy types* to the actual type name to yield the final error message:

```
          ClassCastException Cannot use a map
(clojure.lang.PersistentArrayMap) as a number
                  (java.lang.Number)
```

The main problem one faces when implementing fuzzy types is the problem of determining what the fuzzy type for an arbitrary class should be. Our solution is to look at the class/interfaces that a class extends/implements. These are then compared against a fixed list of class/interface to fuzzy type mappings. If there is only one match, the decision is simple. However, there might be multiple matches. For example, because a Clojure vector can be used as a function (from index to value), it implements both the `clojure.lang.IPersistentVector` interface and the `clojure.lang.IFn` interface. In cases like this, there is usually only one appropriate fuzzy type. To decide which fuzzy type to report, we define a partial order on the classes/interfaces to determine the most relevant fuzzy type. If no fuzzy type name can be generated, we fall back to giving just the class name in the error message.
Occasionally, a class name is the class name of a named function. If we detect that this is the case, we provide the namespaced-name of the function in place of the class name in the resulting error message.

3.3 Clarifying invalid arguments

The problems with Clojure's reporting of type cast errors were universally mitigable with a relatively simple fix. Unfortunately, issues with error reporting resulting from invalid arguments are not as simply soluble. We must handle multiple exception types, and the current error messages do not always provide enough information to improve the error messages.
As an example, consider the code that causes the error message 7:

```
(nth [1 2 3 4] 6)
```

`nth` returns the nth element of some collection. If the index is not in the bounds then `nth` throws an `IndexOutOfBoundsException`. The exception is thrown, but there is no message with the exception, so the message Clojure displays is

```
IndexOutOfBoundsException
```

This is not a lot to work with. It would be desirable to have a message similar to the one that the Java Collections classes throw, such as `ArrayList`. We thus want the error message that Clojure throws to be

```
IndexOutOfBoundsException Index:  6, Size:  4
```

To implement this change, we had to modify Clojure's source code in several locations so that `nth` would throw the appropriate error message. This did not impose a significant runtime cost as we only added a counter to determine the length of a collection if the collection's length cannot be found in constant time.
Fixing the error message is simple if the index given to `nth` is non-negative, as we either know the length of the collection upfront or we find the length by running out of the collection. When the index is negative, things are more difficult. For the collections that come with Clojure, such an index is always invalid. However, since Clojure supports infinite sequences, we might not always be able to find the length of the collection. Therefore, we

report “unknown” as the length of the collection if the length of the collection cannot be determined in constant time. An example of such an error message is error message 9. Other error message of this type can be found in Appendix A.

3.4 Context-dependent Java exceptions

Some error messages are hard to improve upon because they come from deep inside a function’s implementation. We currently do not have a solution for this class of error messages. We instead describe the issues that surround these types of error messages and discuss the trade-offs of different approaches.
As an example of a context-dependent error message, consider error message 13. The code that generates the error message is

```
(into {} (range 10))
```

This code fragment is trying to add the integers 0-9 to an empty map. This operation generates an error as an integer is not a valid form of a map entry, nor can an integer be treated as a sequence of map entries. The error message that this code currently generates is

```
IllegalArgumentException Don't know how to create ISeq from:
                        java.lang.Long
```

The call to `into` above is equivalent to

```
(reduce conj {} (range 10))
```

where `reduce` is a fold operation, an operation that iterates over the collection in order to create a return value, and `conj` is the function that will try to add the integer to the map. During the reduction, the equivalent of the call that causes the exception is

```
(conj {} 0)
```

Past this point, the code that executes is the Java code in Clojure’s implementation, namely the `cons` method in the `clojure.lang.APersistentMap` abstract class. After the `cons` method determines that `0` is not a valid form of a map entry, it then tries to turn `0` into a sequence. It is at this point that the exception occurs.
Due to the number of factors that inform this error, it is difficult to derive an appropriate error message. In addition, the layers of abstraction raise the question of how to assign “blame” in the error message. Referring to the Java code in the implementation would not be very useful, as it is an implementation detail and usually not directly invoked by the programmer. Referring to `reduce` in the error message also does not make sense because it is not `reduce` that caught the error in the code; `reduce` was just doing its job. There are thus only two functions that it makes sense to refer to in the error message: `conj` and `into`.
There are some reasons why one should refer to `conj`. `conj` is the function where the problem actually occurred, and the documentation for `into` implies that `into` uses `conj`. At the same time, there are reasons why one should refer to `into`. The error occurred

because `into` was given arguments that do not work together, and the code sample that caused the error refers to `into`, and not `conj`.
Complicating matters further are the implications this decision has for the handling of abstractions. If we refer to `conj`, we will break abstractions in the code and expose implementation details when generating error messages. On the other hand, referring to `into` means that we will be trying to find the right separation point between abstractions after the error occurred. For instance, what if this `into` call was part of another function? In that case, it could be argued that the real issue then lies with the function that made the bad `into` call. But continuing this logic would lead to always blaming the top-level function that is being run, which would not constitute useful error reporting.

4 Conclusions and future work

Given that our research represents the first systematic attempt to analyze and improve Clojure's error message system, there are still many basic issues to address. While we have addressed and successfully mitigated problems with type casting and invalid argument error reporting, we have yet to develop a way to deal with context-dependent error messages in a useful manner. Clojure's hosted nature makes this problem more difficult, as one must explain errors coming from the host without depending on the context of the host to make sense of them. Future efforts must be spent solving this problem, further improving the error reporting of the Clojure programming language, and developing good basic principles for the improvement of error message usability in general. To this end, we plan to make our error message system available to Clojure users and study the quality of their experiences with it.

References

[1] CULPEPPER, R., AND FELLEISEN, M. Fortifying macros. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2010), ICFP '10, ACM, pp. 235–246.

[2] GONCALVES, E. On clojure error messages. `http://carpathia.blogspot.com/2010/05/on-clojure-error-messages.html`, May 2010.

[3] HALLOWAY, S. *Programming Clojure*, 1st ed. Pragmatic Bookshelf, 2009.

[4] TRAVER, V. J. On compiler error messages: What they say and what they mean. *Advances in Human-Computer Interaction 2010* (2010), 1–26.

A Exceptions

A.1 Type errors

Error message 1
Code:
`(+ {:a 1, :b 2} 3)`
Explanation:
`clojure.lang.PersistentArrayMap` cannot be cast to `java.lang.Number`
Old Message:
`ClassCastException` `{:a 1, :b 2}` is a map, not a number
New Message:
`ClassCastException` Cannot use a map (`clojure.lang.PersistentArrayMap`) as a number (`java.lang.Number`)

Error message 2
Code:
`(+ 5 "foo")`
Explanation:
`java.lang.String` cannot be cast to `java.lang.Number`
Old Message:
`ClassCastException` `"foo"` is not a number
New Message:
`ClassCastException` Cannot use a string (`java.lang.String`) as a number (`java.lang.Number`)

Error message 3
Code:
`(5 3)`
Explanation:
The code tried to use 5 as a function
Old Message:
`ClassCastException` `java.lang.Long` cannot be cast to `clojure.lang.IFn`
New Message:
`ClassCastException` Cannot use a number (`java.lang.Long`) as a function (`clojure.lang.IFn`)

Error message 4
Code:
`(inc num)`
Explanation:
`clojure.core$num` cannot be cast to `java.lang.Number`
Old Message:
`ClassCastException` `num` is a function
New Message:
`ClassCastException` Cannot use a function (`clojure.core/num`) as a number (`java.lang.Number`)

Error message 5
Code:
`(map inc [1 2 :3 4])`
Explanation:
A keyword is not a number
Old Message:
`ClassCastException` `clojure.lang.Keyword` cannot be cast to `java.lang.Number`
New Message:
`ClassCastException` Cannot use a keyword (`clojure.lang.Keyword`) as a number (`java.lang.Number`)

Error message 6
Code:
`(int [1])`
Explanation:
A vector is not a number
Old Message:
`ClassCastException` `clojure.lang.PersistentVector` cannot be cast to `java.lang.Character`
New Message:
`ClassCastException` Cannot use a vector (`clojure.lang.PersistentVector`) as a number (`java.lang.Number`)

A.2 Invalid arguments

Error message 7
Code:
`(nth [1 2 3 4] 6)`
Explanation:
The index 6 is out of bounds
Old Message:
`IndexOutOfBoundsException`
New Message:
`IndexOutOfBoundsException` Index: 6, Size: 4

Error message 8
Code:
`(nth [1 2 3] -5)`
Explanation:
The index -5 is out of bounds
Old Message:
`IndexOutOfBoundsException`
New Message:
`IndexOutOfBoundsException` Index: -5, Size: 4

Error message 9
Code:
`(nth (range) -5)`
Explanation:
The index -5 is out of bounds
Old Message:
`IndexOutOfBoundsException`
New Message:
`IndexOutOfBoundsException` Index: -5, Size: unknown

Error message 10
Code:
`{:a 1, :b}`
Explanation:
There is no value for the key `:b`
Old Message:
`ArrayIndexOutOfBoundsException` 3
New Message:
`IllegalArgumentException` Cannot create a map from an odd number of items: 3

Error message 11
Code:
`(apply sorted-map-by #(+ 1 %) [1 2 3 4])`
Explanation:
`#(+ 1 %)` should take two arguments, not one
Old Message:
`ArityException` Wrong number of args (2) passed to: `user$eval16$fn`
New Message:
`IllegalArgumentException` Function is not a valid `java.util.Comparator`

Error message 12
Code:
`(apply sorted-map-by (constantly "hello") [1 2 3 4])`
Explanation:
The comparator should return a number, not a string
Old Message:
`ClassCastException` `java.lang.String` cannot be cast to `java.lang.Number`
New Message:
`IllegalArgumentException` Function is not a valid `java.util.Comparator`

A.3 Context-dependent error messages

Error message 13
Code:
`(into {} (range 10))`
Explanation:
`into` needs a sequence of pairs, not a sequence of numbers
Old Message:
`IllegalArgumentException` Don't know how to create `ISeq` from: `java.lang.Long`
New Message:
A new message has not been created yet