Work In Progress: an Empirical Study of Static Typing in Ruby

Mark T. Daly   Vibha Sazawal   Jeffrey S. Foster
University of Maryland, College Park
{mdaly,vibha,jfoster}@cs.umd.edu

Abstract

In this paper, we present an empirical pilot study of four skilled programmers as they develop programs in Ruby, a popular, dynamically typed, object-oriented scripting language. Our study compares programmer behavior under the standard Ruby interpreter versus using Diamondback Ruby (DRuby), which adds static type inference to Ruby. The aim of our study is to understand whether DRuby's static typing is beneficial to programmers. We found that DRuby's warnings rarely provided information about potential errors not already evident from Ruby's own error messages or from presumed prior knowledge. We hypothesize that programmers have ways of reasoning about types that compensate for the lack of static type information, possibly limiting DRuby's usefulness when used on small programs.

1. Introduction

In recent years, there has been considerable interest in lightweight, general-purpose scripting languages. The exact definition of a scripting language is debatable, but one common feature is *dynamic typing*, in which types are strongly enforced but are not checked until the last possible moment during execution. While dynamic typing is flexible and admits a range of interesting and useful coding patterns, it also risks runtime type errors that could be found proactively by a static type system.

In this paper, we describe an in-lab pilot study of programmer use of types in Ruby, an object-oriented, dynamically typed scripting language. We collected data from four skilled programmers as they completed two small programming tasks in Ruby, one using the standard Ruby interpreter, with dynamic typing, and one using Diamondback Ruby (DRuby), which adds static type inference to Ruby [Furr et al. 2009b,a]. DRuby includes type system features like intersection and union types, parametric polymorphism, structural object types, and optional and variable type lists for method signatures. Prior experience shows that DRuby finds errors in a range of existing Ruby programs, when used by DRuby's authors [Furr et al. 2009b,a]. In our study, we aim to understand whether DRuby's static type system actually helps typical Ruby programmers find and fix errors—if not, why not, and if so, how could we improve DRuby's type system to better serve programmers' needs?

Based on qualitative analysis of participant experiences, we made three tentative findings. First, using an open coding technique [Strauss 1987] to classify DRuby error messages produced during participant trials, we found that under 20% of DRuby's error messages were *informative*. Second, in interviews, participants reported that they did use types as part of their reasoning process during development. These two findings seem to be at odds—DRuby's type error messages are not helpful, but types themselves are important. We believe the disparity can be explained by the small scale of the programming task studied: In small, single-author programs, developers can rely on their own memory and naming conventions to track type information.

Finally, we found that all four participants used IRB, the interactive Ruby shell, to explore ideas during development. IRB even served as a documentation source for method names and return types. This suggests that DRuby should also offer an interactive interface, possibly by integrating DRuby with IRB.

2. Background

The Ruby Programming Language Ruby is a strongly typed, object oriented programming language whose concise syntax and flexible, dynamic type system is intended to provide programmers with the latitude to write programs in whatever way they wish. The language's creator asserts that, "I want to make Ruby users free. I want to give them the freedom to choose... if there is a better way among many alternatives, I want to encourage that way by making it comfortable" [Venners 2003]. The success of Ruby is reflected both by the considerable community of users and enthusiasts who contribute to its evolution, and by its use as a host language for the popular Ruby on Rails web framework.

In our experience, Ruby's plasticity is a double-edged sword: because the language's interpreter performs few

PLATEAU '09 Workshop at Onward! 26 October 2009. Orlando, Florida, USA.
ACM .

static checks, extensive testing may be required to find programming errors. As a result, practices such as "test-driven development" [Beck 1999], wherein tests are written even *before* code is written, are popular in the Ruby community. However, as is well-known, testing is necessarily incomplete, which raises the question, could static analysis benefit Ruby programmers?

Diamondback Ruby: Static Type Inference for Ruby Diamondback Ruby (DRuby) is a static type inference system for the Ruby programming language. DRuby has been used to identify type errors in a number of small, existing Ruby programs, with most programs requiring little modification to be compatible with DRuby's analysis [Furr et al. 2009b]. Subsequent work showed how to scale up DRuby to highly dynamic language constructs and larger programs [Furr et al. 2009a]. While these results are promising, it is difficult to predict how and to what end such a type inference system would be used by programmers. We would like to know, does static type inference present information to programmers that helps them correct errors?

3. Method

Our pilot study of programmer behavior consists of an inductive, two-treatment, repeated measures experiment in which participants solve short Ruby programming exercises. The experimental conditions differ in either applying DRuby or not to the participant's source code each time the participant executes the Ruby interpreter.

Tasks We gave the participants two programming exercises: writing a simplified sudoku solver and writing a maze solver. The former problem is a simplification of problem found on the *Ruby Quiz* website [Gray 2008], and the latter was inspired by the "Gang of Four" design patterns book [Gamma et al. 1995]. The exercises are of approximately equal difficulty and have little overlap, to discourage direct code reuse. We also aimed for exercises that are complex enough to warrant using DRuby while remaining solvable within the experimental protocol's time limits.

Protocol Each exercise consists of three components: a textual problem description, starter code, and set of test cases that target the top-level API participants are expected to implement. The problem description defines the programming task, describes any data input and output formats, and provides pseudo-code for algorithms that the participant can use to solve the problem. The starter code consists of any boilerplate we expected not to vary among solutions. For the sudoku solver, we supplied a method to iterate over the cells of a serialized sudoku puzzle and a method to calculate the grid region of a given cell. For the maze solver, we supplied methods to parse textual maze definitions. Finally, the test cases give participants a way to run their solution, and we deem solutions that pass all test cases to be correct. The programming task packages as presented to participants are available online at `http://www.cs.umd.edu/~mdaly/druby_pilot_problems.tar.gz`.

Experimental Setup We recruited four participants from a local Ruby users group. We targeted participants in this way because the behavior of novices may not reflect that of more practiced participants [Mayer 1981], and we expected users group members to be comfortable with Ruby. All participants indicated that they are quite familiar with the Ruby programming language. Participants may or may not have used DRuby prior to this study—previous experience (or lack thereof) was not a prerequisite for participation.

The pilot was conducted in a laboratory setting. Participants selected their first exercise, and were allowed as much time as they wished to digest the textual problem description. After participants indicated they were done reading the problem description, we allowed them one hour of programming time. (Participants were not shown the time, but some chose to monitor it themselves.)

Participants used a single development platform, with a standard keyboard, mouse, and monitor. We configured the platform with Emacs, Vi, and TextMate, which are popular with Ruby on Rails programmers [Bray 2007]. We also gave participants access to the Ruby core documentation and, except for the first participant, the Internet. (The first participant was not given Internet access to prevent the use of existing code in this study, but we quickly realized this was a mistake. Participants who followed were simply asked not to copy existing solutions.)

The use of DRuby was randomly selected for one of each participant's problems. DRuby was enabled automatically for executions of the selected problem, requiring no additional action by participants. DRuby is a drop-in replacement for the Ruby interpreter that first performs static type inference and then runs the standard interpreter.

We recorded screenshots and audio as participants worked. Whenever a participant ran the Ruby interpreter or DRuby, we made a snapshot of the source code and the output of the interpreter or DRuby. (The first participant's output had to be recreated after the study due to issues with our software.)

At the end of the first problem, participants could take a break at their discretion before beginning the other problem. After the two programming periods had finished, we asked participants to complete a short questionnaire, and we also interviewed the participants informally to assess their reaction to DRuby and to the study as a whole.

4. Participant Experiences

Next we discuss the experiences of our four participants, ordered chronologically.

Participant A Participant *A* indicated that he is equally comfortable with Java and Ruby, and is somewhat familiar with the C programming language.

Participant *A* only finished about a quarter of each exercise. The reason is that he was given *no* starter source code, which is what our protocol originally stipulated. As a result, participant *A* barely got to write the portions of his solution that might have lead to type errors, rendering use of DRuby mostly moot. We added starter code for subsequent participants to address this issue, and the other participants were able to nearly complete all their exercises.

Although participant *A* made very little use of DRuby, he did take preemptive action to avoid a type error in which data read from a file must be explicitly coerced into an integer. He identified this particular error without the assistance of the Ruby interpreter, DRuby, or any other automated means. Screen recordings show him adding an integer constant to certain variables that store data from a file, and then writing explicit coercions for these variables at an earlier point in the program. While the arithmetic operation may have lead him to find this potential error, we do not know for sure.

In our interview, participant *A* discussed the role of types in Ruby programming. He indicated that he maintains imprecise mental knowledge of types: *"I know that types are there. When I read in a file, I know that I've got a string; [when] I split on newline, then I know I've got an array, so in my head... I have usually an idea that I've got an enumerable. I'm not sure if it's an array or something else..."*

Participant B Participant *B* said that he is quite familiar with Ruby, but is most familiar with Java. He indicated that he is as familiar with C# and Groovy (a Java-like dynamic language) as he is with Ruby, and somewhat so with Python.

Participant *B* encountered some bugs in our data collection tools during the course of writing his solutions. This interfered with some executions of his program and caused him to make some unnecessary edits to his code. Using his feedback, most of these issues were corrected.

In his first programming problem, participant *B* encountered a significant type error: where participant *A* caught the necessary string-to-integer coercion step early on, participant *B* did not discover this until the Ruby interpreter raised a "TypeError" exception. After encountering this error, participant *B* spent several minutes making extensive edits to his program to solve the problem. This occurred during the trial that did not use DRuby.

During his interview, participant *B* described how he continuously keeps type information in mind to supplement the lack of type annotations in Ruby source code. Discussing his experience with Ruby, he stated, *"...with a dynamic language, I'm just kind of subconsciously* always *thinking about types."* In contrast, he explained that, *"...when I'm coding Java, I'm not even thinking about [types], because it's already done for me. So if I make a mistake, the compiler is doing that for me. So, I'm almost consciously just not caring, and so I don't really worry about keeping that stuff in mind..."*

Participant C Participant *C* stated that he is equally familiar with Java and Ruby. Additionally, he indicated some familiarity with C++ and Scheme.

Participant *C* encountered a type error in which he used a single-element array where a value was expected as the contents of an array cell. This error resulted in a failure of the supplied test case, which rather confusingly reported that "4 != 4." The strange error message occurred because of the default printing method for Ruby arrays: a single-element array is printed as just the element itself, without brackets (unlike multi-element arrays). This error happened during the trial that did not use DRuby, and required several minutes of the participant's time to diagnose and correct.

In his interview, participant *C* said that he might not benefit from the sort of error messages he saw reported by DRuby. He explained, *"I do find myself...regularly checking the types of objects to make decisions, usually when I'm making rendering decisions, 'how do I want to render this,' where knowing 'does this object respond to a certain method' [i.e., what DRuby could report] isn't really what I need to know."* This position is understandable given that many of the DRuby error messages he saw concerned calls to methods that had not been implemented. Moreover, when asked to consider shortcomings of Ruby's standard dynamic type system, he stated that he has not been disappointed: *"...my expectations were lowered and then adjusted, so it was more, 'don't rely on types.'"*

Participant D Unlike the previous participants, participant *D* said he is equally familiar with Perl, C, and Ruby. He also said that he is quite familiar with C++ and moderately so with Haskell.

Interestingly, participant *D* made few, if any, type errors during his development. With the exception of some (Ruby interpreter) errors due to uninitialized hash table cells, none of the error messages produced by participant *D*'s test executions indicated a type mismatch.

In his interview, participant *D* said that the relatively small scope of the solutions he was asked to write made DRuby's error messages rather ineffectual. He said, *"it would usually be faster to run the test suite without running the static checks, because [the programming challenges] were such small programs,"* and that, *"[DRuby] usually told me things that I already knew, like...I hadn't implemented a particular method yet—I knew I hadn't implemented a particular method yet, but wanted to see the initialization go through."*

5. Results

Because of the limited number of participants in our study, it is difficult to come to definitive conclusions. Nevertheless, we were able to inductively formulate several tentative hypotheses using the data we gathered; we expect to investigate these more fully in future studies.

DRuby's Error Messages: Correct but Not Informative To analyze the DRuby error messages that our participants received, we assigned each error message to one or more categories using open coding. Open coding is a method of inducing hypotheses from qualitative data by comparing fragments of data with each other, assigning attributes (called codes) to each fragment, and grouping fragments together into categories based on those codes [Strauss 1987].

To categorize the DRuby error messages, we considered each message with respect to other simultaneously reported messages, any warnings produced by the Ruby interpreter in the same execution of the participant's program, any code changes made by the participant since the last execution of the program, and all DRuby messages that preceded it.

We ended up with seven primary codes for DRuby error messages: a) *Duplicate*: multiple messages representing the same error for different sites in a single execution; b) *Intentional*: the result of an intentional edit with obvious consequences; c) *Expected*: seen in an earlier execution or expected from starting conditions; d) *Identical*: same as a warning message reported by Ruby; e) *Additional*: not reported by Ruby for that execution; f) *New*: previously unreported error; g) *Recurrence*: previously seen message from a reintroduced bug. In the example output:

```
[ERROR] instance Sudoku does not support \
  methods print_puzzle
  in method call s.print_puzzle
  at ./sudoku.rb:33
  in typing ::Sudoku.new
  at ./sudoku.rb:32

[ERROR] wrong arity to function, got exactly \
  1 arguments, expected no arguments
  in solving method: initialize
  in typing ::Sudoku.new
  at ./sudoku.rb:32

[ERROR] wrong arity to function, got exactly \
  1 arguments, expected no arguments
  in solving method: initialize
  in typing ::Sudoku.new
  at ./sudoku.rb:34

sudoku.rb:32:in 'initialize': wrong number of \
  arguments (1 for 0) (ArgumentError)
        from sudoku.rb:32:in 'new'
        from sudoku.rb:32
```

the first DRuby message (prefixed with [ERROR]) is coded as *Additional*. The second and third would be coded as *Identical* since the same warning is reported by Ruby (the final message), and the third as *Duplicate* because it is the same as the second. If these errors occurred in a previous execution or if this was one of the first executions of the program (when the programmer has not had a chance to write any methods yet), these would also be marked *Expected*; otherwise the first error would be coded as *Recurrence* or *New* depending on whether or not it had occurred and been fixed before. The *Duplicate*, *Expected*, and *Identical* codes were applied to messages very frequently. The *Additional* and *New* codes were applied less frequently, and the *Intentional* and *Recurrence* codes were applied to very few messages.

These codes were grouped into categories representing whether a message did or did not provide information to the programmer in excess of what they could be expected to already know or could have obtained through using Ruby alone. Messages were assigned to one primary category, either *Informative* or *Not Informative*, based on the codes they had received: a message was assigned to *Informative* if it had been given at least one of *Additional*, *New*, or *Recurrence* exclusively; otherwise, it was assigned to *Not Informative*.

The primary theme that emerged from our analysis is that DRuby did not reliably contribute much useful information. While a limited number of error messages were classified as *Informative* by our open coding scheme, the majority were not: excluding data from participants *A* and *B*, who experienced problems with the protocol and data capture software that were already discussed, 13.4% of error messages were classified as *Informative*, and only 20% executions where DRuby reported at least one error contained any *Informative* error messages. These percentages are lower if participants *A* and *B*'s data is included.

That said, none of the messages produced by DRuby were incorrect, and it may be that DRuby is useful for larger projects but not for the small programs in our study. One of DRuby's key advantages over standard testing is that it analyzes all code paths, including obscure ones—of which there may be few in small programs with straightforward control flow. Further investigation will be required, however, before we can make this claim with confidence.

Programmer Conventions as Type Annotations The coding technique applied to DRuby's error messages shows that DRuby did not report much useful information. However, our interviews indicated that types are part of participants' reasoning processes. This disparity is troubling, as we would expect programmers to find type errors more easily with the aid of DRuby. There may be, however, other mechanisms at work that prevent type errors in the first place.

While it is always wise to give methods and arguments names that indicate what they mean or do, they can also be used to encode type information. In this example (from participant code, as are all those that follow), the parameter names indicate their types directly:

```
  def validate_digits(array, str)
```

This is not the only way that participants encoded type data into their method definitions. The type signature of the method:

```
  def set_value_at(x,y,value)
```

is also partly obvious in the context of a program that uses a two-dimensional grid. It is a reasonable guess that x and y are integer coordinates; `value` could have any type, but the programmer would probably be able to easily remember its specific type. Other method definitions in the same program include:

```
def get_value_at(x,y)
def row_values(x)
def col_values(y)
def grid_values(cx,cy)
```

Again, arguments that include `x` and `y` in their names are probably integers. Each method's name contains `value` or `values`, and so will probably return the same type of objects that `set_value_at` takes as an argument. These are not precise type signatures, but programmers do not necessarily need precision when dealing with small programs.

Another convention also appeared in participants' code:

```
def open?(sym)
```

The use of a question mark at the end of methods does not change a method's behavior, but is a general Ruby convention: "Methods that act as queries are often named with a trailing `?`, such as `instance_of?`" [Thomas et al. 2004]. In this case, we are asking if an object is open or not for some symbol, and so expect `open?` to return a boolean.

All of our participants wrote method definitions that appear to specify some amount of argument or return type data. Ad-hoc conventions may have helped to limit type errors, but further study would be required to know for certain.

Sources of Type Information: Ruby as its Own Reference Several participants indicated that they rely on their own memory to compensate for the lack of explicit type information in Ruby. Moreover, in reviewing interview tapes and screen recordings, we found that participants used several resources when their memory was insufficient: They gathered information from the Ruby Core documentation, the Internet at large, the `ri` utility (a command-line tool for accessing Ruby documentation), and IRB, the interactive Ruby shell. Based on our recordings, IRB is by far the preferred method for exploring features of Ruby; participant *A* even said in his interview that he uses reflection in Ruby to look up method names. (The "methods" method can be invoked on a class to get a list of its methods.) IRB was the only information resource employed by all four participants; one participant used IRB for everything from experimenting with certain Ruby constructs, to manually loading and executing portions of his program, to looking up a particular method's return type.

If programmers prefer IRB over other forms of Ruby reference material, then tools like DRuby may be more effective if they provide interactive documentation as well. In its current form, DRuby provides type documentation through rich type annotations written in comments. A more effective form of documentation may integrate annotations into the output produced by IRB (as is done, for example, by OCaml's interactive shell).

6. Threats to Validity

One key threat to the validity of our study is the scale of the programs written by the participants. In our experience, many Ruby programs are created by writing larger, reusable libraries and then writing small main programs; our study captures only the latter. Additionally, one very common use of Ruby is to write programs in Ruby on Rails, which is not included in our study—Rails code is not statically analyzable by DRuby by itself [An et al. 2009].

Another uncontrolled variable is the effect of DRuby itself on participants' workflow. During this study, DRuby typically took about 80 times longer to analyze and run participants' code than when run just under Ruby. This delay in execution was clearly noticeable, and may have motivated participants to test their programs less frequently when using DRuby—this change in debugging practices may have affected their development processes, though we cannot know for certain.

7. Related Work

While a great deal of research has been conducted regarding human factors in software development, little work has focused specifically on the effect of type systems on programmer behavior. Gannon [1977] studied the error rates in solutions to programming problems written in untyped vs. statically typed variants of a programming language. However, in Gannon's study, participants were graduate and advanced undergraduate students, while our participants were recruited from a users group for the language being studied. Additionally, type systems in particular and programming languages in general have evolved a great deal since Gannon's work.

Ng Cheong Vee et al. [2005] explored the effect of various kinds of compiler error messages on both novice and "mature" students using Eiffel, categorizing errors based on log data collected during the study. Yang et al. [2000] investigated manual type checking practices in Standard ML, but used existing code containing errors rather than having participants write their own programs. Recently, Hanenberg [2009] completed preliminary research on the effect of typed vs. untyped variants of a novel language, finding that programmers worked faster in the untyped version.

While DRuby was selected for this research, other static type inference and checking systems for dynamically typed languages exist. Morrison [2006] developed a type inference approach that is used by the RadRails IDE for Ruby on Rails. Because this inference system is built into a specific IDE, however, it was not well suited to our study. Several type inference systems for Python have been developed by Aycock [2000], Cannon [2005], and Salib [2004]; additionally, Ancona et al. [2007] have created a statically typed subset of Python that can be compiled to CLI or JVM bytecode. Similar systems, such as CMUCL [MacLachlan 1992] and SBCL [SBCL 2008], have been developed for Lisp.

8. Future Work

Our pilot study allowed us to gain insight into the practices of Ruby programmers and to refine our experimental protocol. There are several interesting directions for future work.

An alternate approach to our study would be to conduct surveys. Ayewah and Pugh [2008] surveyed users of FindBugs, a static analysis tool for Java, to gain an understanding of how it is used in practice. They also have investigated the use of FindBugs in industrial settings [Ayewah et al. 2007]. However, similar studies with DRuby (or other static type systems for dynamically typed languages) would first require a sizable user-base, which we do not believe currently exists.

Another direction would be to scale up our study to larger programs. We could ask participants to identify errors in existing software projects of varying size and complexity, and observe whether DRuby helps them find and fix bugs.

A study of programmers working as a team might also be interesting. Code changes by multiple developers may cause inconsistencies in their respective understandings of a program's types, creating opportunities for type errors. This might also allow us to investigate the use of conventions as annotations, as the type information encoded by one programmer may not be obvious to another.

Acknowledgments

We wish to thank Michael Hicks and the anonymous reviewers for their helpful comments on earlier versions of this paper, Mike Furr and Elnatan Reisner for assisting in the testing of our protocol, and our study participants. This research was supported in part by DARPA ODOD.HR00110810073.

References

J. D. An, A. Chaudhuri, and J. S. Foster. Static Typing for Ruby on Rails. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand, Nov. 2009. To appear.

D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. RPython: Reconciling Dynamically and Statically Typed OO Languages. In *DLS*, 2007.

J. Aycock. Aggressive Type Inference. In *Proceedings of the 8th International Python Conference*, pages 11–20, 2000.

N. Ayewah and W. Pugh. A report on a survey and study of static analysis users. In *DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems*, pages 1–5, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-051-7. doi: http://doi.acm.org/10.1145/1390817.1390819.

N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-595-3. doi: http://doi.acm.org/10.1145/1251535.1251536.

K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, October 1999.

T. Bray. Ruby survey results, November 2007. `http://www.tbray.org/ongoing/When/200x/2007/11/26/Ruby-Tool-Survey`.

B. Cannon. Localized Type Inference of Atomic Types in Python. Master's thesis, California Polytechnic State University, San Luis Obispo, 2005.

M. Furr, J.-h. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proceedings of the twenty fourth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2009a.

M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. Static Type Inference for Ruby. In *OOPS Track, SAC*, 2009b.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

J. D. Gannon. An experimental evaluation of data type conventions. *Commun. ACM*, 20(8):584–595, 1977. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359763.359800.

J. E. Gray. Ruby quiz, February 2008. `http://www.rubyquiz.com/`.

S. Hanenberg. What is the impact of type systems on programming time? – first empirical results. Proceedings of the 2009 Workshop on Evaluation and Usability of Programming Languages and Tools, October 2009.

R. A. MacLachlan. The python compiler for cmu common lisp. In *ACM conference on LISP and functional programming*, pages 235–246, New York, NY, USA, 1992. ISBN 0-89791-481-3. doi: http://doi.acm.org/10.1145/141471.141558.

R. E. Mayer. The psychology of how novices learn computer programming. *ACM Comput. Surv.*, 13(1):121–141, 1981. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/356835.356841.

J. Morrison. Type Inference in Ruby. Google Summer of Code Project, 2006.

M. Ng Cheong Vee, B. Meyer, and K. L. Mannock. Empirical study of novice errors and error paths. Unpublished technical report, 2005.

M. Salib. Starkiller: A Static Type Inferencer and Compiler for Python. Master's thesis, MIT, 2004.

SBCL 2008. Steel Bank Common Lisp, 2008. `http://www.sbcl.org/`.

A. L. Strauss. *Qualitative Analysis for Social Scientists*. Cambridge University Press, Cambridge, United Kingdom, 1987.

D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2nd edition, 2004.

B. Venners. The Philosophy of Ruby: A Conversation with Yukihiro Matsumoto, Part I, Sept. 2003. `http://www.artima.com/intv/rubyP.html`.

J. Yang, G. Michaelson, and P. Trinder. How do people check polymorphic types? Proceedings of the 12th Workshop on the Psychology of Programming, April 2000.