Autocompletion for the Rest of Us

Nick Shelley

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Jay McCarthy, Chair
Bryan Morse
Dennis Ng

Department of Computer Science

Brigham Young University

August 2014

Copyright © 2014 Nick Shelley

All Rights Reserved

ABSTRACT

Autocompletion for the Rest of Us

Nick Shelley
Department of Computer Science, BYU
Master of Science

Code completion systems act both as a way to decrease typing and as a way to increase awareness. The former is typically done by completing known variable or function names, while the latter is done by providing a list of possible completions or by providing documentation. Static type information makes these goals possible and feasible for qualifying languages, so most work in this area is focused on improving the order of results or trimming less-valuable results. It follows that almost all validation techniques for this work have focused on proving how well a completion system can put a desired result at the top of the list. However, because of the lack of static type information in dynamically-typed languages, achieving the aforementioned goals is much harder, and many of the completion suggestions may even result in compile-time or runtime crashes. Unfortunately, of the work done on creating completers for these languages, little validation work has been done, making it hard to determine what improvements can be made. This thesis will provide two validation techniques that will provide information both on how well completion suggestions are ordered and also which completion suggestions result in errors. This information will be used to guide the development and evolution of a completion system for the Racket programming language.

Keywords: code completion, auto-complete, autocomplete

ACKNOWLEDGMENTS

Thanks to Dr. McCarthy for his support and patience.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The main purpose of autocompletion is to improve programmer productivity. This has mainly been accomplished in two ways. The first way is to reduce typing by predicting which token(s) should be inserted at a given position, sometimes replacing the partially-typed token. Incidentally, this also encourages more descriptive variable names since longer names can be completed automatically. The second way is to provide convenient documentation to the programmer. For example, a list of currently valid (i.e. in scope, correctly typed, etc.) substitutions may be provided given the current cursor position. Also, a function's signature and purpose can be displayed next to the typed function name. This project will focus on the first way of completing textual tokens rather than providing convenient documentation.

Autocompletion for statically-typed languages is relatively simple to do naively. Since type information is available statically, it is easy to avoid suggesting tokens that will result in compile or runtime errors. Because static type information makes it possible to suggest completions with high assurance of being valid, research and work has focused mainly on presenting these valid results in a way that is more useful to the programmer. For instance, instead of giving all valid results in alphabetical order, results can be omitted or promoted closer to the top of the list based on context, type, or analysis of previous code. Therefore, validation systems have mainly focused on how well a desired token makes it in the top N suggestions.

However, because type information is not known statically in other languages, it is possible and easy to suggest tokens that may result in runtime errors and others that may not

even compile. Other features that make autocompletion hard for certain languages include first-class functions and powerful macro systems. I will now discuss the challenges these features pose in detail.

1.1 Dynamic Typing

Dynamically typed languages do the majority of their type checking at run-time as opposed to compile-time. In dynamic typing, values have types but variables do not. A variable can thus refer to any value of any type at any time.

Because a variable's type can change at any time in the program, the type information can't be used to filter which variables can appear in which contexts. For example, consider the incomplete programs in Figure 1.1. Should an auto-completer give the parameter *a* as an option to fill the hole in `dynamic-parameter.rkt`? With the call (*add2* `5`), the program would have a meaningful result. However, with the call (*add2* `#t`), there would be a run-time error. Since the auto-completer can't know what types of values the parameter *a* will be at run-time, it cannot know whether it is correct to include it in the auto-complete list.

As another example, consider the two holes in `dynamic-return-value.rkt`. The auto-completer needs to make a decision about whether or not to include *num* in the list of suggestions. If *num* filled the first hole, the number `10` would be inserted and the result of the program would be `15`. However, if *num* filled the second hole, the string `"hi"` would be inserted and a run-time error would occur. Although it is easy for us as humans to look at these two function calls and know what they will return, it is hard for a computer to do so without running the program first. We can also imagine more complicated functions which would be hard for both humans and computers to analyze. Since the auto-completer can't even know what type a function will return, it cannot know whether it is correct to include the result of a function in the auto-complete list.

`dynamic-parameter.rkt`

```
#lang racket
(define (add2 a)
  (+ [ ] 2))
```

`dynamic-return-value.rkt`

```
#lang racket
(define (return-something a)
  (if (< a 5) 10 "hi"))

(define num (return-something 2))
(+ [ ] 5)
(set! num (return-something 8))
(+ [ ] 5)
```

Figure 1.1: Problems with Dynamic Typing

1.2 First-class Functions

First-class functions allow functions to be passed around as values. Coupled with the dynamic typing problem, first-class functions make it impossible to know which bindings are functions and which aren't. Ideally, if an auto-completer is invoked where it makes sense to call a function (such as after an opening parenthesis), only functions should appear in the list of suggestions. However, because we can't know which bindings are functions, we must consider all bindings. For example, the program `function-parameter.rkt` in Figure 1.2 contains two holes, one where a boolean should go, and another where a function should go. Because passing + to *fun* is just as valid as passing #t or 3, we can't know if it is correct to include the parameter f in either of those holes.

Even if we could know or assume that a function is being called, we cannot know which function a variable is bound to. This makes bringing up information about the function, such as a parameter list, impossible. For example, the program `function-return.rkt` in the same figure contains two holes. In the first instance we would want the auto-completer to bring up the possible parameters to the + function that will be returned, which are z .... In the second instance we would want the auto-completer to bring up *make-vector*'s

`function-parameter.rkt`

```
#lang racket
(define (fun f)
  (when [ ] ([ ] 3 2))
```

`function-return.rkt`

```
#lang racket
(define (return-fun a)
  (if (< a 5) + make-vector))

((return-fun 2) [ ])
((return-fun 7) [ ])
```

Figure 1.2: Problems with First-class Functions

possible parameters, which are *size* $[v]$. However, because we don't know what function will be returned, we cannot bring up the correct parameters.

1.3 Powerful Macro System

Racket macros are essentially an API to add compiler extensions. All macros compile down to a small set of core language features, potentially going through intermediate transformations when other macros are used in macro definitions. Most of the features available in #lang **racket** are macro definitions. For example, the **and** and **or** constructs are actually macros that compile to the core **let** and **if** constructs. Macros such as **or** can be used to define other macros, which can be used to define yet more macros. A program is said to be fully expanded when all macros have been recursively transformed until only core language constructs remain.

Syntax objects are an important part of the Racket macro system. Syntax objects basically wrap source code with lexical and location information. For example, a syntax object for the symbol x could have lexical information describing which bindings are visible to x.

Racket macros are transformers that take a syntax object and return a syntax object, or in other words, they take source code and return source code. The syntax object returned can do anything it wants with the syntax object passed in, or it could ignore its input and

return an arbitrary syntax object. In other words, Racket macros can do arbitrary things to the program.

Figure 1.3 shows macros that can cause problems for an auto-completer. In `binding.rkt`, the *not-define* macro introduces a binding for the variable x, breaking its promise not to define anything. It does this by extracting the value from the syntax object passed in and creating a new syntax object that defines x as this value. However, by examining the program, an auto-completer has no indication that x has been defined anywhere. In particular, at the hole in the + application in `binding.rkt`, the auto-completer is not aware that x is an available binding.

In `context.rkt`, x is bound to `5` within the **let**. By examining the source code, an auto-completer could give x as an option at the hole within the **let**. However, in this case the addition expression is passed to the f macro. This macro essentially replaces the context of the expression passed in with the context of *anchor*. Since *anchor* was defined at the top-level, the expression returned from f will only have access to top-level bindings. This can cause two problems. As we see in `context.rkt`, x is bound to `#f` at the top level, so filling the hole with x would cause run-time error to occur. Even an auto-completer that was smart enough to get around the dynamic typing problem by only including bindings of the correct type would be fooled by this scenario. Thus the first problem is essentially a complication of the dynamic typing problem. However, if we remove the definition of x at the top-level, then there would be no binding for x available to the expression, which would cause a compile-time error.

A possible work-around for macro-related problems could be to fully expand the program before invoking the auto-completer. However, it is currently impossible to expand incomplete programs. Some possible ways around this problem could be to judiciously choose points in the program at which to attempt expansion or to reduce the current program to a complete expandable program. My project will help to show how useful these approximations will be.

`binding.rkt`

```
#lang racket
(define-syntax (not-define stx)
  (syntax-case stx ()
    [(_ v)
     (with-syntax ([id (datum->syntax stx 'x)])
       (syntax
        (define id v)))]))

(not-define 6)
(+ [ ] 2)
```

`context.rkt`

```
#lang racket
(require (for-syntax syntax/strip-co
(define-for-syntax anchor #'here)
(define-syntax (f stx)
  (syntax-case stx ()
    [(_ e)
     (replace-context anchor #'e)]))

(define x #f)
(let ([x 5])
  (f (+ [ ] 2)))
```

Figure 1.3: Problems with Macros

As shown, these language features make it especially hard to implement completion systems for such languages. Because of this, work in this area has focused on providing suggestions in a similar way to static completers. Because validation efforts for such completers are minimum, improvements to such systems relies on user complaints, and implementing new systems correctly is hard at best.

Chapter 2

Related Work

Some commercial and research completion systems have been implemented for dynamically-typed languages such as PHP and JavaScript [1, 5, 7, 8]. These systems use knowledge about the structure of the language to complete common constructs such as functions, objects, and variables. Dynamic information is either ignored or else semantic analysis and type inference is used to statically determine as much information as possible. None of these systems provide or document any sort of testing or validation framework.

For traditional completion systems, many possible improvements have been explored over the traditional alphabetical sorting of all possible suggestions. [6] uses intimate knowledge about the Java type system, library and API functions, example code, and the current context of the program to suggest ways to get from one type to another type through intermediate steps. [9] sorts by Java type starting with methods provided from the declared type and then up the type hierarchy until the base Object class. It also allows for custom filters to be specified by the programmer. [3] takes abbreviated input instead of prefixes and uses a hidden markov model to determine what was meant. For example, the expression "ch.opn(n)" would turn into "chooser.showOpenDialog(null)." [10] tracks all changes to the codebase and prioritizes suggestions based on how recently methods and objects have been changed. [2] uses existing code (such as the Eclipse code base) as training data to implement three different completion systems. The first orders suggestions based on how frequently they are used. The second uses limited context information to create association rules that govern suggestion order. For instance, one rule might be that after objects are created, suggest setters on

those objects. The third gathers context and uses a modification of the k-nearest-neighbor algorithm to order suggestions based on the nearest snippets found in the training data.

Most validation techniques ran their systems on modified existing code and ranked the results in some way. [3] ran their system on 3000 lines of code from 6 open source projects that were turned into acronym-like abbreviations. They measured how many lines could be completed and how many of those were in the top-N (i.e. top-1 or top-5) suggested completions. [4] ran their completion system at every Swing method in several open source projects. They ranked how high in the suggestions list the actual method appeared and added all rankings together to get an overall ranking for the system. [10] ran their completion system between every recorded change operation (defined as a change in the AST) on a project of theirs called SpyWare. They then recorded what percentage of actual methods were in the top-N suggestions. Lastly, [2] used the Eclipse code base and randomly split the code into 90% training data and 10% test data. They then took out half of the Swing method calls in the test data and ran their completion system in each hole. They measured recall, which is how often a used method was in the suggestion list, and precision, which is what fraction of suggested methods were actually used. They then combined these two measures into an F1-measure to give an overall grade to a system.

The lack of testing done on dynamic auto-completers helped me realize the need for a system that does what my second validation method will do. The testing done on most static auto-completers inspired the first validation method of random testing on existing code. The metrics they used are all similar and will help me define my metrics in a way that works best for this project. Lastly, although a lot of improvements were type-system specific, some could be used as ideas for completion approximation methods in my thesis.

Chapter 3

Implementation

There are two major metrics to be considered when testing code completion. The first is how well a completer can get expected results near the top of the list. The second is how well a completer can keep failing results out of the list. I wrote a separate testing framework for each of these metrics, both of which use common base functionality.

3.1 Foundation

There are two main ways to tell whether code completion meets users' needs. One is to do user studies or get feedback from users, while another is to measure completion results against existing code. This testing framework takes the second approach.

In order to test against existing source, the source file must be tokenized. One way to do this is just to delimit on whitespace. However, since it is uncommon for data such as numbers and strings to use code completion, we decided to use the language grammar to only test valid identifiers.

I used Racket's built-in reader to tokenize existing Racket source. I take the results of the reader and store each identifier along with source location in a word structure. This identifier+location structure is what is used in all of the testing.

Since testing every token of every source file can take a very long time when the amount of source code is large, we used randomized testing to save time while still collecting representative data. To do this, I collect all of the identifiers from a file as described above and then randomly choose a percentage of those tokens to test against.

When an identifier from the original source is tested, the source file can be modified in several ways. We chose two simple ways to modify the source. The first way is to simply remove the token from the source file, which represents modifying or maintaining existing source code. The second way is to truncate the source file from the chosen identifier, which represents writing new source code.

3.2 Ranker

For a given token in a source file, the ranker will modify the file at that position, either by removing or truncating as described above, and then invoke an auto-completer to get a list of completion results. It then compares the list of results to the original token, which represents the programmer's intent. If the original token is found in the first five results, its rank is recorded. It is also noted whether the original token is found later in the list or not at all.

This method of testing completion results allows changes affecting the number and location of results to be checked quantifiably and consistently.

3.3 Checker

Similar to the ranker, the checker asks an auto-completer for a list of completions from a modified source file. However, instead of checking against the original token, the original token is replaced by one of the completion results, and the resulting program is compiled and run. If the program runs to completion, the completion results is recorded as a success. If the program fails to compile or terminates for any reason, the type of error is recorded.

This method allows auto-completers to test how top completion results fail for different methods of completion. With this information, targeted changes can help reduce certain types of failures.

Chapter 4

Verification

Stuff

4.1 Textual Heuristics

Textual heuristics uses source code tokenization to determine which completions to suggest. Suggestions are prioritized based on proximity.

4.1.1 Method Implementation

This implementation is fairly straightforward. We just separate based on the language's defined delimiters by processing the whole text file with the following regular expression: [^\s()[]",'‘#|\;]+

The resulting strings and positions are combined into the word struct. When the completer is invoked at a specific position, these results are prioritized based on the following sort order:

s(w1, w2) = |(w1 position) - (completer position)| < |(w2 position) - (completer position)|

4.1.2 Ranker Results

Stuff

4.1.3 Checker Results

Stuff

4.2 Structural Heuristics

The structure of the language can give good hints about how to filter and prioritize completion suggestions. The two ways we did this for Racket are by nesting level and by keywords and position.

4.2.1 Method Implementation

For nesting level, we took each word returned by the textual heuristics method and attached nesting level information to it. We did this by starting at the beginning of the file and keeping track of how many parentheses and brackets had passed. The nesting level was incremented for each opening parenthesis or bracket and decremented for each closing.

This nesting information was then used to prioritize results. Suggestions were ordered based on the difference in nesting level between the current position at completion invocation and each suggestion.

For language keywords, a regular expression was used to find all function names based on the define and let keywords. We then determined whether the position of the completer invocation was in function application position. In Racket this is a simple check: if the position is directly after an opening parenthesis, it is in function application position. If this was the case, all function names we found previously were prioritized ahead of all other tokens.

4.2.2 Ranker Results

Stuff

4.2.3 Checker Results

Stuff

4.3 Macro Heuristics

Stuff

4.3.1 Method Implementation

Stuff

4.3.2 Ranker Results

Stuff

4.3.3 Checker Results

Stuff

4.4 Macro Analysis

Stuff

4.4.1 Method Implementation

Stuff

4.4.2 Ranker Results

Stuff

4.4.3 Checker Results

Stuff

Chapter 5

Conclusions

Stuff

References

[1] Syntactic and semantic prediction in dynamic languages. In Roger Lee and Naohiro Ishii, editors, *Software Engineering Research, Management and Applications 2009*, volume 253 of *Studies in Computational Intelligence*, pages 59–70. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-05440-2. URL `http://dx.doi.org/10.1007/978-3-642-05441-9_6`.

[2] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2. doi: 10.1145/1595696.1595728. URL `http://doi.acm.org/10.1145/1595696.1595728`.

[3] Sangmok Han, David R. Wallace, and Robert C. Miller. Code completion from abbreviated input. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 332–343, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3891-4. doi: 10.1109/ASE.2009.64. URL `http://dx.doi.org/10.1109/ASE.2009.64`.

[4] Daqing Hou and David M. Pletcher. Towards a better code completion system by api grouping, filtering, and popularity-based ranking. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 26–30, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-974-9. doi: 10.1145/1808920.1808926. URL `http://doi.acm.org/10.1145/1808920.1808926`.

[5] jcx.software. Vs.php. http://www.jcxsoftware.com/vs.php.

[6] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. *SIGPLAN Not.*, 40:48–61, June 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1064978.1065018. URL `http://doi.acm.org/10.1145/1064978.1065018`.

[7] Microsoft. Jscript intellisense. http://msdn.microsoft.com/en-us/library/bb385682.aspx.

[8] Jakub Misek and Filip Zavoral. Mapping of dynamic language constructs into static abstract syntax trees. In *Proceedings of the 2010 IEEE/ACIS 9th International Conference on Computer and Information Science*, ICIS '10, pages 625–630, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4147-1. doi: 10.1109/ICIS.2010.100. URL `http://dx.doi.org/10.1109/ICIS.2010.100`.

[9] David M. Pletcher and Daqing Hou. Bcc: Enhancing code completion for better api usability. In *ICSM*, pages 393–394. IEEE, 2009. URL `http://dblp.uni-trier.de/db/conf/icsm/icsm2009.html#PletcherH09`.

[10] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2187-9. doi: http://dx.doi.org/10.1109/ASE.2008.42. URL `http://dx.doi.org/10.1109/ASE.2008.42`.