Macho: Programming With Man Pages

Anthony Cozzie, Murph Finnicum, and Samuel T. King
University of Illinois

Abstract

Despite years of work on programming languages, programming is still slow and error-prone. In this paper we describe Macho, a system which combines a natural language parser, a database of code, and an automated debugger to write simple programs from natural language and examples of their correct execution. Adding examples to natural language makes it easier for Macho to actually generate a correct program, because it can test its candidate solutions and fix simple errors. Macho is able to synthesize basic versions of six out of nine small coreutils from short natural language descriptions based on their man pages and sample runs.

1 Introduction

Programming is hard. Because computers can only execute simple instructions, the programmer must spell out the application's behavior in excruciating detail. Because computers slavishly follow their instructions, any trivial error will result in a crash or, worse, a security exploit. Together they make computer code difficult and time consuming to write, read, and debug.

Programmers write software the same way they do everything else: by imitating other people. The first response to a new problem is often to google it, and ideally find code snippets or examples of library calls. The programmer then combines these chunks of code, writes some test cases, and makes small changes to the program until its output is correct for the inputs he has considered.

Software engineering researchers have developed techniques to help automate each of these parts of the programming process. Code search tools scan through databases of source code to find code samples related to programmer queries. For example, SNIFF [2] uses source code comments to help find snippets of code, and Prospector [4] finds library calls that convert from one language type to another. Automated debugging tools not only help find problems [6], but sometimes even suggest solutions [7]. For example, recent work by Weimer *et al.* [5], describes how to use genetic programming algorithms to modify buggy source code automatically until the modified programs pass a set of test cases.

Although these techniques do save time, the programmer is still responsible for selecting code snippets, arranging them into a program, and debugging the result. In this paper we describe Macho, a system that generates simple Java programs from a combination of natural language, examples (unit tests), and a large repository of Java source code (mostly from Sourceforge projects). It contains four subsystems: a natural language parser that maps English into database queries, a large database that maps programmer abstractions to snippets of Java code, a stitcher that combines code snippets in "reasonable" ways, and an automated debugger that tests the resulting candidate programs against the examples and makes simple fixes automatically.

Because database search and automated debugging are still hard problems with immature tools, Macho's abilities are correspondingly basic. Our current version of Macho was able to synthesize simple versions (no options, one or two arguments) of various Unix core utilities from simple natural language specifications and examples of correct behavior, including versions of ls, pwd, cat, cp, sort, and grep. Macho was unable to generate correct solutions for wget, head, and uniq. Macho is still under construction, but it has already provided us with several interesting results.

Macho is a remarkably simple attack on an extraordinarily difficult task. Natural language understanding is considered one of the hardest problems in Artificial Intelligence with a huge body of current research. Generalizing from examples is similarly difficult. And even once a computer system "understands" the problem it still must actually write suitable Java code.

Our key insight is that natural language and examples have considerable synergy. Macho has a fighting chance to generate correct programs because each component can partially correct for the mistakes of the others. For example, a database query will return many possible results, most of which will be incorrect, but by leveraging the type system the stitcher can eliminate many unlikely solutions. Even more importantly, the test cases allow Macho to partially detour around the difficult problem of natural language processing. Modern machine learning techniques provide probabilistic answers, whether the question is the meaning of a piece of natural language or the best sample function in the database to use. Backed by its automated debugger, Macho can afford to try multiple solutions.

In addition, combining examples and natural language greatly reduces their ambiguity: the set of programs that satisfies both the natural language and the test cases is much smaller than the sets that satisfy each input individually, although there are some exceptions: Macho found it surprisingly easy to synthesize cat from a unit test using the empty files it used for generating ls. However, we found that most of the time a program that passed even one reasonable test case would be correct. Together natural language and examples form a fairly concrete specification.

2 Architecture

Macho's workflow mirrors a human programmer. It maps the natural language to implied computation, maps those abstractions to concrete Java code, combines the code chunks into a candidate solution, and finally debugs the resulting program. The goal of each subsystem is therefore to minimize the amount of brute force and thereby synthesize the largest possible programs.

2.1 Natural Language Parser

Our natural language parsing subsystem attempts to extract implied chunks of computation and the data flow between them from the words and phrases it receives, and encode that knowledge for the database. Usually the structure of the sentence can be directly transformed to requested computation: verbs imply action, nouns imply objects, and two nouns linked by a preposition imply some sort of conversion code. This mapping is conceptually similar to previous work [1], but Macho's database "understands" a much larger number of concepts, including abbreviations. In order to handle these more varied sentences, we began with an off-the-shelf system provided by the University of Illinois Cognitive Computation group to tag individual words with their part of speech (noun, verb, adjective, etc.) and to split sentences apart into smaller phrases.

Our main problem was fixing the errors of the parser, which was trained on a standard corpus of newspaper articles, not jargon filled man pages. For example, 'file' is usually a verb, like "the SEC filed charges against Enron today." and print is often a noun, e.g., "Their foul prints will not soon be cleansed from the financial system.". These kinds of errors were quite common.

To help detect what words were intended to act as actions, we build a graph of prepositions linking the objects in a sentence together into a tree. A traversal of this tree reveals the relationship between the nouns at its leaves. When we find words that are not linked to the rest of the sentence by this graph, we can guess that they are misclassified verbs. The parser also provides some hints as to likely control flow. For example, plural adjective or adverbial phrases often imply a filter operation that is implemented as an if statement. The description of grep contains 'lines matching a pattern' which implies only some lines will be used.

2.2 Database

As the subsystem that maps natural language abstractions to concrete Java code, the database is the engine that powers Macho. When the database can suggest reasonable code chunks, the stitching can usually find a correct solution, but when the database fails the space of candidate programs is simply too large to succeed by flailing randomly.

Our original plan was to use Google Code, but we almost immediately dismissed it as completely inadequate. Google Code indexes a huge number of files, but it appears to only perform keyword search on the raw text of the source files, which we found to be inadequate for our problem. Instead, we developed our own database for Macho.

Our first step was to obtain a data set of about 200,000 Java files from open source projects and compile them using a special version of javac that we modified to emit abstract syntax trees. We compiled rather than parsed because we wanted exact global locations for each function call, and because we didn't want to reuse broken code. Since open source programmers are not exactly paragons of code maintenance, only about half of our source files compiled successfully.

Our database returns candidate methods based on input and output variables, e.g. the query $directory \rightarrow files$ would return all functions called with an input variable named directory and assigned to a variable named files. This nicely captured the different abstractions that different programmers used to represent code, which is important because functions have only one name. The problem with this approach is that many things aren't usually implemented as functions. Higher level concepts

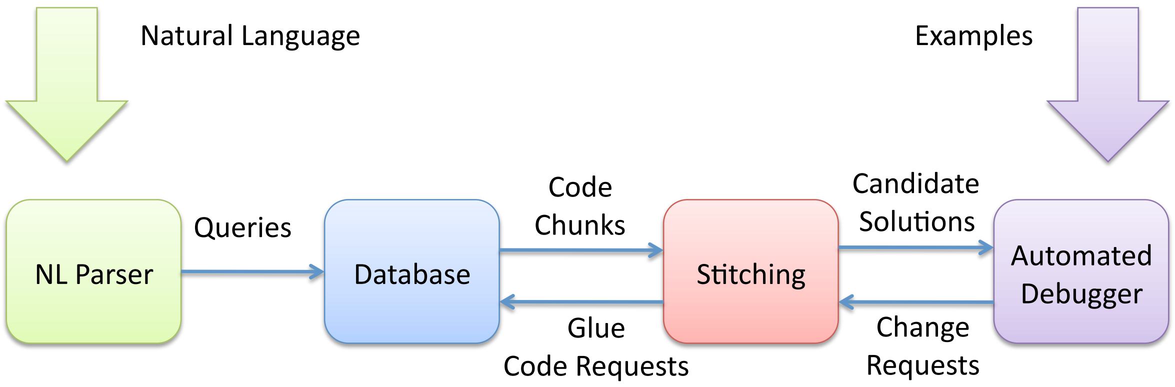

Figure 1: Macho workflow

like ignore, first, or adjacent usually appear as operations or even control flow. Often they have no input variables or are only tagged in the comments.

2.3 Stitching

Macho's stitching subsystem combines results from database queries into candidate programs. Its main guide is the type system; two expressions can be linked by a variable if the output type of one matches the input type of the other. If the types don't match, the stitcher will query the database for common chunks of code that were used to convert between those types.

Macho also generates a small amount of control flow. If statements are generated only from hints by the natural language parser and the synthesizer. Map loops are generated when suggested by the type system. Macho tries to limit control flow generation because it swiftly increases the solution space; an upstream chunk may be placed in any block above the downstream chunk.

The most difficult part of stitching is keeping track of the data flow between expressions in the presence of control flow. The natural language gives a great deal of information for how information is supposed to flow from one chunk to another; previous natural language programming systems generated code without any search at all.

2.4 Automated debugger

Macho's automated debugging subsystem attempts to debug candidate programs. This type of automated debugging is potentially extremely difficult, but many of the automatically generated candidate programs will have utterly obvious errors that can be fixed easily. The primary difference between stitching and automated debugging is that debugging is dynamic rather than static and has access to the behavior of the program. Currently the automated debugger runs the candidate in a sandbox and performs a diff between the output of the candidate and the unit test and classifies the candidate into one of five simple cases: exception thrown (try to insert an if block around the offending statement), a superset of correct output (insert if blocks around the offending print), garbage (try the next program), a subset of correct output (try adding a few prints), or, in the best case, correct output (declare victory).

These components have synergy beyond simply correcting mistakes. For example, our automated debugger leverages the database to suggest changes to buggy programs. When it is faced with a potential solution for ls which incorrectly prints hidden files, the debugger queries the database for commonly used functions of *java.io.File* which could be used in an if statement to restrict the obstreperous print. This simple probabilistic model allows it to try the *isHidden* method even though it is not used elsewhere in the candidate solution.

Although the automated debugging seems superficially simple, it actually solves a very difficult problem of library combination. Macho's database finds candidate functions entirely by name, which may be unrelated to their purpose. Running the code allows the debugger to eliminate these imposter functions.

3 Evaluation

Objectively evaluating Macho is very difficult. There is no standard test suite where we can benchmark our results against other systems, and using the language from the man pages directly is almost impossible. Consider the byzantine man page description for wget:

> GNU Wget is a free utility for non-interactive download of files from the Web. It supports HTTP, HTTPS, and FTP protocols, as well as retrieval through HTTP proxies.

Program	Result	Input	Notes
pwd	success	Print the current working directory.	Difficult as there is no input.
pwd	success	Print the user directory.	CWD = “user.dir” in Java.
pwd	success	Print the current directory.	Abbreviation!
pwd	fail	Print the working directory.	Breaks NLP for arcane reasons.
pwd	fail	Show the current working directory.	Database entries for show are mostly graphics.
cat	success	Print the lines of a file.	Vanilla.
cat	success	Read a file.	Print is synthesized.
cat	fail	Display the contents of a file.	Database entries for contents are mostly graphics.
cat	fail	Print a file	Solutions print the file name.
sort	success	Sort the lines of a file.	Print is synthesized.
sort	success	Sort a file by line.	
sort	fail	Sort a file.	Insufficiently precise specification.
sort	fail	Sort the contents of a file	Database entries for contents are mostly graphics.
grep	success	Print the lines in a file matching a pattern.	Solutions using both JavaLib and GNU regexes.
grep	fail	Find a pattern in the lines of a file.	Correct except for if statement linking test and print.
grep	fail	Search file for a pattern.	Poor resiliency for function names.
ls	success	Print the names of files in a directory. Sort the names.	
ls	success	Print the contents of a folder. Sort the names.	
ls	fail	Print the names of the entries in a directory.	Entries to names fails.
ls	fail	Print the files in a directory.	Does not synthesize sort.
cp	success	Copy src file to dest file.	Programmer abbreviation!
cp	success	Copy file to file.	Ugly but Macho needs to know there are two inputs.
cp	fail	Duplicate file to file.	No candidate in database.
wget	fail	Download file.	Candidates have extra functionality.
wget	fail	Open network connection. Download file.	Macho can’t create buffer transfer loop.
head	fail	Print the first ten lines of a file.	’First’ is incomprehensible.
uniq	fail	Print a file. Ignore adjacent lines.	’Ignore’ and ’adjacent’ don’t map to libraries.
perl	fail	The answer to life, the universe, and everything.	Seems to work, but it’s still running.

Figure 2: Macho’s results for generating select core utils. This figure shows the results for pwd, cat, sort, grep, ls, cp, wget, head, and uniq, and the natural language input we used for each of these programs.

Giving out partial credit is also difficult. Some of Macho’s solutions are very close but not byte identical, but automatically determining whether or not an output is sufficiently close to the test case is approximately as hard as generating the program, an artificial version of the Dunning-Kruger effect. Under these circumstances we decided to try to pick an interesting set of natural language inputs right on the border of Macho’s capabilities and use our best judgement when the test cases were “close”.

Macho succeeded in generating simple versions of six out of nine coreutils - pwd, cat, sort, grep, cp, and ls - and failed to synthesize wget, head, and uniq. For each core utility, we targeted its default behavior: no options and the minimum number of arguments possible. Since we had the programs available anyway, we used them to generate our unit tests. All of the programs had only one short test and the results are shown in Figure 2.

4 Lessons Learned

4.1 The Database is King

Although most of the programs Macho writes are 10-15 lines or less, there are a *lot* of potential 10-line Java programs. Brute force really does not get very far - the ability of the database to select reasonable pieces from the natural language heuristics is absolutely critical. In general, when the stitching failed, it was often reasonable to think of a hack, or a simple fix, or just let it run a little longer, but when the database failed Macho had no hope of ever generating a correct solution. Improving Macho will require a superior database above everything else.

4.2 Pure NLP is Bad

Programming with natural language is generally considered a bad idea because specifying details gradually mutates the natural language into a wordy version of Visual Basic. Consider a natural language spec for ls:

```
Take the path "/home/zerocool/"
If the path is a file, print
it.  Otherwise get the list of
files in the directory.  Sort the
result alphabetically.  Go over
the result from the beginning to
the end:  If the current element’s
filename does not begin with ".",
print it.
```

which is our best guess for the input required for Pegasus [3]; it is obvious why most programmers would prefer to use Python instead. Instead, a Macho programmer can specify the basic task very simply:

```
Print the names of files in a
directory.  Sort the names.
```

Even an almost trivial program like this leaves many details unspecified: should the sort be alphabetically by filename, size, file extension, or date? Should the program print the full path, the relative path, or just the name of the files? Does "files" include subdirectories or hidden files? All of these questions are easily cleared up by an example of correct operation. Such examples not only have a higher information density than tedious pages of pseudocode or UML, but they also reduce the workload of the programmer by allowing him to think about one case at a time, rather than all possible cases. In other words, examples allow a user to be concrete without being formal.

4.3 Interactive Programming is the Answer

A traditional programmer must write code that satisfies all possible inputs his program will encounter, while a Macho Programmer can consider each input individually. Macho therefore not only saves the programmer the work of writing code but also frees the programmer from difficult formal reasoning.

Ideally, however, the programmer would only be required to verify, not generate, concrete values. In this rosy scenario the programmer would input natural language and the system would offer a set of alternatives. The programmer could then reject incorrect cases, or suggest modifications, until eventually a correct program is negotiated. This is important because programming is not simply the act of transferring a mental vision into machine code. In reality, the requirements are fuzzy. Some things are more important than others, and still others can be waived or changed if they are difficult to implement. Interactive programming allows the programmer to take the path of least resistance to a satisfactory program.

Of course, this also requires considerably more accurate program synthesis from pure natural language, as well as much better understanding of general concepts, which no one really knows how to do at the moment.

5 Conclusions

In this paper we have discussed Macho, a system that synthesizes programs from a combination of natural language, unit tests, and a large database of source code samples. A few of our technical findings are that the natural language can give implicit hints about the control flow in a program, variable names contain useful information about the functionality of code, and the automatic debugger can use the database to add new code to a candidate solution.

Macho is a simple proof of concept system, not yet directly useful for most programmers, but it can still synthesize basic versions of six small coreutils. By improving the source code database we believe that Macho can be a practical system for helping programmers.

References

[1] A. W. Biermann and B. W. Ballard. Toward natural language computation. *Comput. Linguist.*, 6(2):71–86, 1980.

[2] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for Java using free-form queries. In *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 385–400, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] R. Knöll and M. Mezini. Pegasus: first steps toward a naturalistic programming language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 542–559, New York, NY, USA, 2006. ACM.

[4] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI '05: Proceedings of the 2005 Conference on Programming Language Design and Implementation*, pages 48–61, New York, NY, USA, 2005. ACM.

[5] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[6] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.

[7] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 272–281, New York, NY, USA, 2006. ACM.