Macho II: Even More Macho

Anthony Cozzie and Samuel T. King
University of Illinois at Urbana-Champaign
{acozzie2,kingst}@illinois.edu

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.

Software engineering and programming languages researchers have developed techniques and tools 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 [5] finds library calls that convert from one language type to another. Automated debugging tools not only help find problems [7], but sometimes even suggest solutions [8]. For example, recent work by Weimer *et al.* [6], describes how to use genetic programming algorithms to modify buggy source code automatically until the modified programs pass a set of test cases.

Unfortunately this impressive arsenal spends a great deal of time on the bench, as most programmers prefer to go into battle armed only with a copy of Emacs, a few Google searches, and their own personal wetware. Part of the problem is the Sisyphean effort required to keep up with technology in the 21st century: new languages, tools, and libraries appear every few years. Part of the problem is that most of the tools still require manual intervention: either they are based on probabilistic machine learning methods, which will not always return the correct answer, or they do not have the whole picture of the program, part of which is locked in the programmer's mind. If a code search tool suggests using Function A to solve a particular problem, the programmer must read the man page for Function A and see if it will really work. If Coverity suggests that a certain input might crash the program, the programmer must carefully reason it through.

Our system, Macho, attempts to solve both of these problems simultaneously, by simply running several tools at once and checking their results against an example of correct output (effectively a short unit test). This reduces all tools to a simple, standard interface (what should my program do in case x?) but more importantly it is an *end to end check on multiple tools*. If Macho can generate correct output for the examples, there is a high probability that all of the tools did their job correctly, which means that the programmer does not have to check each one individually.

If we are going to check the results of multiple tools, we must start with a fairly high-level input, and the most logical candidate is natural language. Natural language is familiar to everyone, and is flexible enough to work at many levels of abstraction. However, programming in natural language is considered somewhat taboo since the 80s, and for good reasons. When the input is sufficiently abstract, it becomes ambiguous. This is difficult for both the programming system, which must make many decisions, and for the user, who does not know which choices were made [3]. So natural language programming systems are traditionally verbose versions of Visual Basic, with syntax characters replaced by wordy equivalents [1, 4], while true natural language understanding is considered one of the hardest problems in Artificial Intelligence.

Macho avoids this dilemma by also requesting an example of correct output. Our key insight is that those choices that the system must make to resolve the ambiguity in natural language are exactly the same ones that the programmer would make while guiding the tools towards the correct answer. We found that examples and natural language have tremendous synergy: the natural language provides moderate information over the entire input space while the example provides precise information over a tiny fraction of the input space. A program that satisfies both is much more likely to be correct than one that satisfies either individually. The tools themselves also synergize, because their decisions are not independent: which function we select to do subtask A will depend on which function we select to do subtask B, and so on.

2. Design

Programming from natural language and examples is rather tricky. Our workshop paper at Hotos in May explained our first cut at this problem, but our intial version simply wasn't capable enough to be considered a real system, even by us. We are currently working on a complete redesign based on our experiences. Our goal this time around is to build a real system (although it would probably on be practical for some users in some scenarios) that we will hopefully be able to demo during the poster session.

Macho I taught us that learning even simple programming patterns from a set of source files in an unsupervised manner is really hard. We thought that we could mine all sorts of things like 'before you can read a file, you have to open it', and it turned out to just be very difficult. Many things were not labeled - it's just not possible to learn that you can ignore something by encasing it with an if statement when that if statement is only labelled with a noisy comment. Of course, we tried to use the database for good reasons: we did not (and still do not) want to replicate the expert systems of the 80s which were build purely with hand-written rules. Such systems are usually extremely brittle.

So in Macho II we are defining the patterns manually, but using machine learning to glue them together, select functions from

the database, and to rank the resulting programs. Currently we are building a probabilistic model of Java code using techniques from natural language processing. In the same way that NLP uses the surrounding words to predict whether a given word is a verb or a noun, we can use the input variables and context functions (both names and types) to predict what function was used. We can build similar models for variable names, loops and conditionals, dataflow, and so on, eventually ending up with a complete generative model for a program. Our hope is that this gives us the robustness of probabilistic methods while still allowing us to fix random problems manually.

We also have a potential answer to the question 'What language should Macho program in?'. We mentioned this point in passing in our workshop paper, but we assumed it was rather academic since most of our training data would be in Java, C, C++, or C#, which are all fairly similar languages. However, it occurred to us that our model was really learning to make predictions based on operations, variables, and their interactions, which would allow us to select almost any programming language.

We chose a graph based, dataflow language which we call Spoon (there is no spoon!) for several reasons. First, it allows us to represent the program as it is gradually convered from a grammar parse tree through pseudocode to Java in one representation. Second, it reduces the number of candiate programs by ignoring the order of operations. Third, it allows us to write patterns without worrying about silly Java syntax (these patterns are essentially macros. There is a reason the only languages with good macro support are Lisp variants - its just too obnxious to worry about otherwise). Fourth, it allows us to print pretty graphs of Macho's attempts to generate an interesting solution, which are usually pretty easy to understand.

3. Summary / Q&A

Although it tries to solve the same problem, Macho II is almost completely different under the hood. Sadly, our demo isn't quite ready. What can we say, programming is hard . . .

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] E. W. Dijkstra. On the foolishness of 'natural language programming'. `http://userweb.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD667.html`.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.