Andrew Begel: Research Statement

Computers play many crucial roles in our society, from the desktops in the workplace to the cell phones in our pockets. The software that runs on these systems has grown increasingly large, complex, and prone to bugs. One reason is that the tools used to program these systems have not kept pace with their growth. The tools impede programmers from catching bugs early, communicating program concepts easily and efficiently with colleagues, predicting how a program will behave when used by others, and understanding the history of the software artifact that is being built. Software has simply grown too large and complex for programmers to understand without tool support.

The solution is to lower the programmers' cognitive load, permitting them to concentrate their skills on the problem they are trying to solve, rather than wasting their skills on the rote details of the code. Some programming environments have already begun to incorporate features that realize this solution. For example, many IDEs support program refactorings, which are high-level program transformations that encompass many of the simple tasks performed during routine software maintenance. IDEs also provide dynamic feedback regarding the correctness of authored code without requiring the programmer to invoke the compiler. They offer guidance to proper usage of large, complex APIs without requiring the programmer to read the manual. Runtime-generated code profiles help programmers optimize their programs by identifying performance bottlenecks without requiring the programmer to pore over assembly code and timing measurements. Each of these tools succeeds by combining program analysis with human-computer interaction, utilizing the increased knowledge the computer has about the program and the software development process to enrich the interaction between programmer and computer.

My research at the University of California, Berkeley and at MIT has focused on building these kinds of programming tools and their supporting analyses. Over the past decade, I have worked with programmers of many ages, abilities and physical impairments, designing programming environments to enhance their comprehension, productivity, and performance. The work has led to (a) my dissertation work creating the SPED voice-recognition-based programming environment – enabling those suffering from repetitive strain injuries (and other physical disabilities that prevent them from using a keyboard and mouse) to program efficiently enough to remain competitive in the workforce, (b) the StarLogo programming environment – helping students learn about complex systems modeling through simulation, (c) the BPF+ packet filter language and optimizer – enabling network analysts to use a high-level, expressive predicate language to monitor particular packets from a network stream without sacrificing performance, and (d) a design for a next generation of the HTTP protocol – enabling programmers to describe long-lasting program interfaces in the face of incremental, anarchic evolution of data structures and function signatures. In each of these projects, I have developed new algorithms, architectures, languages, and tools that contribute to the state of the art in research and in practice.

My ultimate goal with this line of research is to understand and improve the programmer-computer interface. Achieving this goal will help make existing programmers more productive, help novices more easily develop expert programming skills, and improve the accessibility of programming environments for people with disabilities. I will take a three-pronged approach: 1) conduct user studies to learn how programmers interact with their programming environments, 2) design tools that will help them overcome the barriers they face in the programming process, and 3) develop novel program analyses and infrastructure that will make these new tools possible. Much of the research I propose to conduct is cross-disciplinary, incorporating aspects from software engineering, programming languages, human-computer interaction, and in the longer term, artificial intelligence and natural language processing. The best setting for my work is in a computer science department that encompasses human-centered computing, encourages interdisciplinary research collaborations, and supports first-class research in software development environments.

Research Contributions

Many programmers suffer from repetitive strain injuries (RSI) and other more severe motor impairments. These individuals cannot easily use a keyboard and a mouse, and have difficulty staying productive in a work environment that all but requires long hours typing code into a computer. My dissertation work helps to lower these barriers by enabling developers to reduce their dependence on typing by using speech. Speech interfaces may help to reduce the onset of RSI among computer users, and at the same time, increase access for those already suffering motor impairments.

My approach to speech-based programming comes from a programming language perspective. By exploiting the domain-specific nature of programming and applying programming language-based analyses to a verbalized form of authoring, editing and navigating through code, I hope to alleviate many of the common problems of voice recognition for programmers. Other work in this area relies heavily on tools provided by speech recognition vendors and rudimentary text-based analyses that do not exploit any knowledge of the programming language or program being written.

Naturally Verbalizable Programs

The goal of my project is to enable input that is natural to speak, yet understandable by the voice recognition systems that must process it. Programming languages have historically been communicated in written form; prior to our work, verbalization of code has been highly ad-hoc and not at all formally defined. We conducted experiments to reveal how programmers speak code. We found that while there does exist a common vernacular among programmers for speaking programs, some aspects of speech present challenges for system understanding. Punctuation is inconsistently verbalized, but generally omitted in certain constructions. Difficulties arise with homophones (words that sound alike but are spelled differently), capitalization of words, and concatenated names. We saw differences between native and non-native English speakers in regards to ambiguous utterances – native speakers use prosody (pitch and pausing) to disambiguate the construct, while non-native English speakers rephrase the construct in other ways. Native English speakers are also better at verbalizing abbreviations and partial words. We observed that programmers tend to identify patterns and describe them, rather than using only their instantiations. Other experiments we have conducted show that conventional searching and navigating by voice requires too much input, is too slow, and incurs more cognitive load than using the keyboard or mouse. Based on these experiments, we have developed Spoken Java, a dialect of Java that is more naturally verbalized by human developers, along with a command and control language designed to enable programmers to find and select pieces of code and modify them in high-level linguistic ways.

Language Analyses for Ambiguous Input Streams

A significant artifact of our work is a software development system that can understand spoken program authoring, editing and navigation, each in isolation and in combination. First, a programmer speaks program code into a microphone. Then, a speech recognizer turns the speech into text that is fed into our analysis system and displayed on a screen. Lexical ambiguities found in spoken input, such as homophonic, misrecognized, unpronounceable, and concatenated words, affect the voice-based programmer's ability to introduce code and use similar sounding words in different contexts. We addressed the problems by extending the Generalized LR (GLR) parsing algorithm to support the three kinds of ambiguities that can arise from a speech-aware lexical analyzer. We then constructed a novel combined lexer and parser generator called Blender to enable language designers to describe programming languages designed for speech. This generator can combine lexical descriptions and grammars from many languages (for instance, from Spoken Java and its associated command language) into a single analysis module. Our lexer and parser produce an ambiguous parse forest of possible interpretations for any given program. We are developing a semantic analysis which uses incrementally updated static semantic information about a program to disambiguate a newly inserted code fragment.

Novel User Interfaces for Manipulating Code

Our studies have shown that browsing and selecting words and phrases with voice recognition is tedious, error-prone, and slow. Navigation commands supplied by voice recognition tools suffer from several flaws: users must speak too many words, make repetitive utterances, and rely on generally poor visual estimation skills. Our replacement for these techniques, a context-sensitive mouse grid, is a program-aware form of direct navigation. It allows programmers to "drill down" hierarchically through their program to select the desired statement or word. Using this tool, programmers can quickly point at a particular program point to indicate where an editing action may take place without having to re-speak potentially difficult-to-verbalize program text. When verbalization is required, for example, when searching for a name in the code, we are developing a phonetics-based search to make it unnecessary for the user to spell. Search results are all presented together, sorted numerically, and shown with surrounding context to enable the user to quickly navigate with a minimum number of utterances.

While we have striven to make Spoken Java as naturally verbalized as possible, there may be situations where the programmer does not know how to express a particular construct. In this case, we are developing a spoken feedback system where any already written construct may be spoken out loud to help teach and reinforce proper input techniques. We are combining this with visual reinforcement of the language used by programmers as they enter each construct into the computer. This will help alleviate any short-term memory problems programmers may suffer when relying exclusively on voice input.

Future Goals

In addition to further development of the ideas studied in my dissertation, much of the research I plan to conduct addresses questions that have arisen during my time in graduate school. (a) How can a computer exploit task-awareness to improve the software development process? (b) Why are programmers notoriously bad at documenting their code? Might enabling programmers to comment by voice alleviate some of the problems? (c) What can we learn from the many years of natural language research that can be used in the more limited domain of program editing to make spoken programming analyses more efficient and expand the range of acceptable input? (d) Compilers and optimizers have become incredibly complex tools; often their output appears mysterious to the programmer. What feedback can be given to programmers to help them understand the compiler's operations?

Data Mining the Program Edit History

The programming process has been extensively studied using audio and video taping of programmers while they work. Programmers have been asked to "think aloud" while they tackle program comprehension, program authoring and program modification tasks. Very few studies directly record all keystrokes and mouse movements because the resulting data is too low-level and difficult to analyze. However, by combining the proper kinds of data mining, log analysis, and search facilities with program analysis, one could design an inference algorithm to learn what high-level programming interpretation should be given to a sequence of keystrokes and mouse movements. This information could be useful both to analyze programmer actions and to help the programmer during the development process itself. An analysis could infer that the programmer was modifying a set of data structures, for example, or was refactoring his or her code in some systematic way. Even without analysis, simply making the program edit history easily visible and searchable could increase collaboration by enabling programmers to understand the design of someone else's code, and learn how it evolved over time with respect to features that were added and bugs that were fixed. The edit history could also be used to automatically generate a first cut at revision control system comments when the programmer wants to commit source code modifications. The system could automatically generate to-do items when it detects the programmer making systematic, but incomplete changes to code, and with a little more analysis, could check off items when it detects that the programmer has finished the task.

Commenting by Voice

It is notoriously difficult to get developers to comment their code, and even more difficult to maintain documentation in the face of changes to the code and its design. Numerous studies have tried to pin this documentation failure on programmers (their education, their workload, their inherent laziness), but another possible explanation is a simple input modality clash. Both comments and code are entered by keyboard – if programmers want to write comments, they have to stop programming for a short time, distracting them from their programming task. If programmers could comment their code orally in an editor which recorded all voice and edit operations, both comments and code could be entered at the same time, minimizing interruption. Audible comments could be attached to the program and saved into the revision control system. Comments could be played back by another developer wishing to learn how the original programmer created the code. They could be transcribed by speech-to-text systems and added as textual comments for fast browsing. Grosz and Shieber, among others, have studied associating comments with program structure, through natural language analysis of the comment text and its position in the edit history. If one were to combine this analysis with knowledge of the syntax and semantics of the code being written, it would be possible to identify references to names in the code, and perhaps to identify references to particular algorithms. If the analysis were robust enough, it might be used to solve one of the harder problems in software maintenance: keeping comments current when the code changes.

Disambiguating Spoken Program Code

Voice-based entry of code introduces many lexical and syntactic ambiguities that cannot be resolved until semantic analysis is run. In the system built for my dissertation, lexical and syntactic analysis phases must generate all possible interpretations of the input in order for semantic analysis to choose the correct one. In some cases, this process may not scale (as natural language researchers discovered about English language analysis in the 1970s). I will pursue ways to use partial parsing (based on my work in program fragment parsing with GLR) and partial semantic analyses to help prune ambiguities as early as possible in the analysis process. Additional techniques can be developed by adapting natural language disambiguation algorithms to the more limited domain of programming languages.

Understanding Compilers and Optimizers

Compilers and optimizers have become very complex over the past decade, incorporating program analyses and optimization techniques that until recently were found solely in research labs. Similarly, it is increasingly difficult to program the platforms that these tools target due to the incorporation of SMT, VLIW, parallel processing and complex memory hierarchies. Some programmers learn how compilers and optimizers typically transform source code into machine code, which is useful for diagnosing performance problems. However, even this knowledge is inadequate when faced with the code that comes out of an optimizing compiler. Prior work on visualization of compiler and optimizer output has concentrated mainly on correlating the debugger's view of the code with the source. This information should be applied to the programmer's view of the code in the program editor, either in source code form, or in cases where the optimization is not representable in source form, in a high-level pseudo code. Programmers have two major questions: 1) what did the optimizer do to my code? and 2) why didn't the optimizer do this optimization here? To answer the first question, the optimizer might report that a) it hoisted this code to that location b) it strength-reduced this mathematical operation, c) it unrolled this loop four times, d) it inlined this function into those call sites, e) these variables are in registers, f) this region is dead code, or g) it vectorized this loop. It is more difficult, but very profitable, to comprehend why the compiler did not perform a particular transformation: a) these two pointers are aliased, so no common subexpression elimination happened here, b) this object-oriented method was not inlined because the receiver class is not statically determinable, c) this code was not hoisted because there is an exceptional control path. Other optimizations may have "almost" been applied, but were abandoned due to one failed predicate – it would be useful to know which predicate prevented the optimization and why.

The Future

This research plan is fairly ambitious, and will likely keep me occupied for the next several years, if not longer. Some of the research involves exploiting research in areas like artificial intelligence, machine learning and natural language processing, in which I do not yet have expertise. My background and the multi-disciplinary approach I have outlined above, combined with a healthy amount of collaboration with colleagues in these research areas will help to achieve my goal of more productive and accessible programming for software developers.