Project Summary: Understanding the POSIX Shell

Michael Greenberg – Pomona College

The POSIX shell—the general, non-proprietary standard command-line interface—is a cornerstone of computer systems. From administration and management to development and deployment. Shell scripts are immensely powerful: administrators can, with a single script, create, modify, delete, and move files not just on a local computer, but on widely distributed servers.

For such a powerful system, the shell is barely understood: its semantics are based on string expansion, and widely considered obtuse and unworthy of attention. But the consequences of not understanding shell scripts can be disastrous: an easy-to-make mistake in a shell script in the popular Steam gaming platform ended up wiping hard drives; it's a common idiom for open source software to install itself by downloading and directly running a shell script. Installing such software, one hesitates... what does that script do, exactly?

This project proposes to develop a semantics for the shell in order to (a) facilitate the development of tools and (b) guide the design of interactive languages. Despite being a 'Turing complete' programming language, the shell's semantics differ markedly from conventional languages. Not only would such a semantics be useful in its own right—helping tool developers interpret the English of the POSIX standard—such a semantics will be interestingly novel. Preliminary work (Greenberg OBT 2017) has shown that PL techniques, like structural operational semantics, can formalize the POSIX standard, clarifying the standard's English with precise mathematics.

Existing tools for understanding the shell work by *syntactic* analysis; semantic analysis tools for shell would be of immense interest in practice. Initial efforts have already involved students in formalizing expansion; miniaturized versions of expansion and other shell features (pipes, redirection) could appear in coursework, as well.

The bulk of the work will build on top of the semantics: tools to analyze shell programs for errors or malicious behaviors. As a basis of these tools, a compiler in the style of λ_{JS} (Guha et al. ECOOP 2010) will translate the shell down to a more analyzable core language, making shell's strange evaluation behavior explicit. We can check our compiler's correctness with both formal models and testing. With the shell's behavior made explicit, conventional tools could analyze translated shell programs. While there will be plenty of challenges in doing analysis in this domain—reasoning about strings in particular, but also understanding the behavior of commonly used programs like `grep` and `find`—translating shell to a standard semantics means not having to devise an abstract interpretation from scratch.

The final bit of proposed work is language design. Shell is unique: string expansion semantics; expressive concurrency primitives; the sliding continuum from line-by-line interactive work to the complex architecture of the Linux boot sequence; scripts developed in a particular way (printing potential commands to the screen before running them); programmers using a vast array of existing programs written in arbitrary languages. Is there a language that keeps some of the shell's defining characteristics while increasing predictability and security? Others have tried to invent new shells (scsh, shill), but they are very far from the interactive POSIX shell. Is there a language that (a) can be used interactively and programmatically and (b) is close enough to, e.g., bash that simple programs still work, but (c) is more predictable and less error-prone? Does shell offer generalizable insights into interactive programming?