Understanding the POSIX Shell as a Programming Language

Michael Greenberg
Pomona College
michael@cs.pomona.edu

We build intricate systems with complex algorithms and invariants, aiming for guarantees of correctness and performance... and then we maintain and deploy these systems with shell scripts! What *are* shell scripts? If the POSIX shell is a programming language, what are its syntax and semantics? Can we apply PL tools to reason about the shell?

In my talk, I will explain why the shell is a good object of study and present my early progress building models to reason about the shell.

1. The POSIX shell

Expert computer users interact with the computer using textual interfaces, also known as *command-line interfaces* or *shells*. Experts prefer shells for their concision and power. Critical system tasks—installation and deployment, automation of routine tasks, maintenance, forensics—are often done via the shell; some tasks can *only* be done via the shell. Code for completely compromising a computer system is called "shellcode" because once an attacker has privileged shell access, they control the system entirely. Shells are synonymous with complete control.

Among shells, the POSIX shell is the *de facto* standard; by lines of code, it is the 9th most commonly used language on github.[1] The POSIX shell standard (IEEE and The Open Group 2016) defines the most widely supported shell, specifying its syntax and semantics in 122 pages, not counting other operations that may or may not be built into the shell (e.g., `kill`, the command for sending signals to processes).

You don't need to look far to find examples of fragile, dangerous, or confusing shell scripts. Perhaps most famous in recent memory is "Steam cleaning":[2] when a shell script in the Steam gaming platform encountered an unexpected filesystem arrangement on a user's PC, it would delete all of a user's files. Despite the untrammeled power of the shell, it's not uncommon to see software installed via `curl http://derp.io/install.sh | bash`, which fetches a URL and feeds its input directly to a shell.[3]

A variety of tools help programmers navigate the POSIX shell's hazards (koalaman 2016; Kamara 2016; Weidmann 2016). Unfortunately, existing static analyses work only at a shallow, syntactic level; existing dynamic analyses only offer limited guarantees. In terms of academic attention, D'Antoni et al. use machine learning to repair commands, but offer no semantic insights (D'Antoni et al. 2016). Mazurak and Zdancewic gave a static analysis for some of bash's expansion's with an eye towards security (Mazurak and Zdancewic 2007). To my knowledge, Mazurak and Zdancewic offer the only formal semantic understanding of a shell in the literature.

[1] `http://githut.info/`, as of November 8th, 2016.

[2] `http://www.theregister.co.uk/2015/01/17/scary_code_of_the_week_steam_cleans_linux_pcs/`

[3] `https://twitter.com/SwiftOnSecurity/status/756182802165489665`

Programming languages researchers should study the shell as a programming language. Computer users have much to gain from increased tool support for the shell; programming languages researches have much to learn, as the POSIX shell has several distinctive features.

2. Distinctive features

Three features distinguish the shell: its evaluation model, its facility for controlling concurrent processes, and its natural transition from command-at-a-time interactivity to automating batch tasks.

2.1 Expansion over evaluation

The POSIX shell diverges from the ordinary evaluation model. Conventional programming languages evaluate an expression by evaluating its parts; the POSIX shell evaluates an expression by *expanding* its parts.

A conventional programming language might run a function like so:

$$\frac{e_1 \ \mathsf{eval}\ \lambda x.e \qquad e_2 \ \mathsf{eval}\ v_2 \qquad e[v_2/x] \ \mathsf{eval}\ v}{e_1\ e_2 \ \mathsf{eval}\ v}$$

To run a function e_1 on an argument e_2, first evaluate e_1 to a function value and e_2 to an argument value, substitute the argument into the body of the function, and then evaluate the substituted body. Similarly, it might evaluate arithmetic like so:

$$\frac{e_1 \ \mathsf{eval}\ n_1 \qquad e_2 \ \mathsf{eval}\ n_2 \qquad n = n_1 + n_2}{e_1 \ \mathsf{plus}\ e_2 \ \mathsf{eval}\ n}$$

That is, the default presumption is that to evaluate an expression, we must evaluate each of its sub-expressions.

In the POSIX shell, functions and commands are evaluated, but their arguments are *expanded*, a process of string substitution. A comparable rule for applying a function in the shell might read:

$$\frac{e_1 \ \mathsf{expand}\ \lambda x.e \qquad e_2 \ \mathsf{expand}\ v_2 \qquad e[v_2/x] \ \mathsf{eval}\ v}{e_1\ e_2 \ \mathsf{eval}\ v}$$

where $e \ \mathsf{expand}\ v$ means that the text of e expands to the value v. The shell has seven stages of expansion (IEEE and The Open Group 2016); for example, the first stage of expansion—tilde expansion—translates `~` to the current user's home path. A rule might look like:

$$\frac{\texttt{LOGNAME} = x}{\texttt{~}\ \mathsf{expand}\ \texttt{/home/}x}$$

While plus might be a function in an conventional language, in the shell arithmetic operations are also done by expansion; arithmetic expansion takes place inside `$((...))` forms and is the fourth stage of expansion. We could model addition like so:

$$\frac{e_1 \ \mathsf{expand}\ s_1 \qquad e_2 \ \mathsf{expand}\ s_2}{\texttt{\$((}e_1 + e_2\texttt{))}\ \mathsf{expand}\ \mathsf{itoa}(\mathsf{atoi}(s_1) + \mathsf{atoi}(s_2))}$$

where atoi converts strings to numbers and itoa converts numbers to strings. While the shell doesn't, by default, evaluate sub-expressions, the user can enclose sub-expressions in `$(...)` or

‘...‘ to have them evaluated using command substitution, the third stage of expansion:

$$\frac{e \text{ eval } v}{\$(e) \text{ expand } v}$$

These tiny caricatures don't do justice to the complexity of the shell; Mazurak and Zdancewic cover a large subset (but not arithmetic expansion) (Mazurak and Zdancewic 2007); in my talk, I will report on my own progress modeling the various stages of expansion.

2.2 Controlling concurrent processes

The POSIX shell has many primitives for managing file descriptors (via `>`, etc.), setting up pipes between processes (via `|`, command substitution, etc.), and controlling concurrent jobs (via `&`, `wait`, etc.). It has succinct syntax and powerful semantics for managing the flow of information. Commands in the shell run in separate memory spaces, using the filesystem as shared "memory". Can we apply the shell's model elsewhere?

2.3 Interactivity, live coding, and automation

I typically use the shell to issue one (possibly compound) command at a time. When facing a repetitive task, I'll try to see if my commands form a pattern, as in:

```
$ grade hw1-latest
$ parse hw1-latest/grades.txt >hw1.csv
$ grade hw2-latest
$ parse hw2-latest/grades.txt >hw2.csv
...
```

Noticing the pattern, I try to automate what I'm doing. But as much as I know the shell, I fear it—so I'll start out with some exploratory debugging.

```
$ for d in *-latest; do
    echo ${d} ${d%-latest}
  done
hw1-latest hw1
hw2-latest hw2
...
```

Once I've found the expansions I'm looking for, I'll actually run the commands I want:

```
$ for d in *-latest; do
    echo ${d%-latest}
    grade ${d}
    parse ${d}/grades.txt >${d%-latest}.csv
  done
hw1
hw2
...
```

The shell encourages a sliding scale from interactivity to batch programming using an iterative "print what you'll do before you do it" programming style. Programming languages are seen as a means for automation, even though conventional languages often perform tasks very differently than humans would. The shell automates human tasks more literally: the `for` loop above runs exactly the commands I would have run manually. By literally "doing what a human would do", the shell is more like a macro system (in the sense of document processing) than a conventional programming language.

Collins et al. looked at "live coding" with an eye towards live development of programmatic musical scores, and there is a sizable literature on the topic (Collins et al. 2003). Shell scripts are also composed in an interactive fashion; what insights might we borrow? Can we apply recent developments in programming by example (Gulwani et al. 2015) and feedback-oriented programming (Chugh et al. 2016) to the shell? Can the shell teach us anything about programming by example? D'Antoni et al. make some progress in this direction, but their synthesis algorithms use examples without any semantic information (D'Antoni et al. 2016). Can we do better if we know something about how the shell works?

3. Opportunities

Tools for making users of shell scripts safer and authors of shell scripts more confident would be a huge boon. Beyond such immediate practical applications, what can we learn from the shell's evaluation and concurrency models? Many shell bugs are due to difficulties in understanding the order of expansion; can we define something *close to* the POSIX shell that doesn't suffer from these confusions? Can we design languages that offer the same ease of transition between interactive use and programming, but in different domains?

4. Proposal

In my talk, I will (a) motivate command-line interfaces and shells as objects of study, (b) argue for the POSIX shell in particular, (c) explain the unique structure of the shell, and (d) report on my early progress understanding the shell and building tools for it.

Acknowledgments

Arjun Guha helped me develop some of my ideas on the topic; Sam Tobin-Hochstadt gave encouragement and pointed me in interesting directions. Two undergraduates at Pomona College have been assisting in the early work: Calvin Aylward and Austin Blatt.

References

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 341–354, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908103. URL `http://doi.acm.org/10.1145/2908080.2908103`.

Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. Live coding in laptop performance. *Organised sound*, 8(3):321–330, 2003.

Loris D'Antoni, Rishabh Singh, and Michael Vaughn. NoFAQ: Synthesizing command repairs from examples. *CoRR*, abs/1608.08219, 2016. URL `http://arxiv.org/abs/1608.08219`.

Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *Commun. ACM*, 58(11):90–99, October 2015. ISSN 0001-0782. doi: 10.1145/2736282. URL `http://doi.acm.org/10.1145/2736282`.

IEEE and The Open Group. *The Open Group Base Specifications Issue 7 (IEEE Std 1003.1-2008)*. IEEE and The Open Group, 2016.

Idan Kamara. explainshell, 2016. URL `http://explainshell.com/`.

koalaman. Shellcheck, 2016. URL `https://github.com/koalaman/shellcheck/`.

Karl Mazurak and Steve Zdancewic. Abash: Finding bugs in bash scripts. In *PLAS*, pages 105–114, 2007. doi: 10.1145/1255329.1255347.

Philipp Emanuel Weidmann. maybe, 2016. URL `https://github.com/p-e-w/maybe`.