1 Introduction

The POSIX shell is the de facto standard for interacting with computer systems, ranging from personal laptops to clusters of many powerful servers [IEEE and The Open Group 2016]. Few languages rival the shell for power, but it has inspired more mockery and revulsion [Garfinkel et al. 1994] than academic attention [D’Antoni et al. 2016; Jeannerod et al. 2017a,b; Mazurak and Zdancewic 2007]. The shell is not taken seriously as a programming language, but it is in fact an extremely powerful DSL for concurrency with strong support for interactivity. In my talk, I will present preliminary work on formalizing the POSIX shell standard. What makes the POSIX shell so good at interactivity [Greenberg 2018] and managing concurrency [Greenberg 2017]? How can we help novices and more experienced users understand the POSIX shell? I will demonstrate a stepper that makes the shell’s obscure semantics observable.

2 The shell is a DSL for interactivity

While the shell is used to power complex systems (e.g., Debian maintainer scripts [Jeannerod et al. 2017b]), it also sees widespread use as an interactive console—the expert’s control hatch. On many systems, there are management tasks that can only be done via the shell. But even when GUI options exist, many users prefer the shell. What makes the shell so good for interactivity?

I believe there are two aspects of the shell’s design that support interactivity: like many other DSLs, the POSIX shell offers extremely concise syntax for common commands (e.g., mv * . c src/); and many of the commonly used shell commands are variadic. For example, the mv command above has a flexible interface, of the form mv file_{1} … file_{n} tgt, where tgt must be a directory if n > 1 but need not be if n = 1. The mv command’s interface is directly supported by the shell’s expansion semantics. In the shell, commands and their arguments are treated ‘stringily’; at runtime, a process called expansion translates strings containing control codes (like the * in the command above, or variables like $PATH) to possibly many arguments. The expansion of * . c, for example, will search the current working directory for files that match (i.e., end in . c), generating and alphabetically sorting an argument for each such file.

Greenberg [2018] has already argued that expansion supports POSIX shell interactivity. For more information on expansion, see either Section 2 of that paper or the POSIX standard [IEEE and The Open Group 2016].

3 The shell is a DSL for concurrency

The POSIX shell succinctly and powerfully manages the flow of information between concurrent processes. It has many primitives for managing file descriptors (> , < , etc.), setting up pipes between processes (|, command substitution, etc.), and controlling concurrent jobs (&, wait, etc.). Each command in the shell runs in a separate memory space, using the filesystem as shared memory. To see how simple it is to work with concurrency in the shell, contrast the following two snippets that run LDA [Blei et al. 2003] to generate topic models for moderately large datasets [Greenberg et al. 2015]:

```bash
for k in ${KS}; do
    lda est 1/50 $(k) \ 
    settings.txt $(ABS) \ 
    seeded $(DIR)/lda${k}
    done
    echo "DONE"
```

(a) Sequential

```bash
for k in ${KS}; do
    lda est 1/50 $(k) \ 
    settings.txt $(ABS) \ 
    seeded $(DIR)/lda${k} &
    wait
    echo "DONE"
```

(b) Concurrent and parallel

In the sequential version (a), we run lda for each value of the parameter k sequentially. When each LDA process completes some hours or days later, it will return control to shell, which will continue the for loop. When the for loop terminates, we announce our completion (echo). Each run of lda is independent—we can speed things up by building models concurrently. The concurrent version (b) makes only two changes. First, we run lda in the ‘background’ (by adding & to the command). Commands run in the background return control to the shell immediately: the for loop will spawn all of the lda processes in the same order—but without waiting. We then use the built-in wait utility to have the shell wait for all of its background processes to terminate. In concurrency terminology, we’ve just used fork/join parallelism: the & is a "fork" and wait is a "join". The speedup from building the models in parallel and concurrently on a multicore machine are appreciable—nearly linear. Beyond the speedup,
analysis focused on finding errors where field splitting produces too many or too few fields. Greenberg [2018] gives an example showing that "field splitting can be controlled at use sites but not at definition sites" (Section 2). Here is another example of field splitting’s trickiness: suppose we are in a directory with two files, a and b and we want to move these files to another directory, ./.other._dir. In typing out the command mv * ./.other._dir, our finger slips and we hit enter before we type the target directory, running the command mv *. What happens? An intuitive mental model of the shell might expect an error: after all, * means "all files" and we never specified a destination. Without a destination, mv doesn’t know what to do, and should produce an error. Unfortunately, that’s not what happens. In fact, the shell expands * to the two files present, a and b, in that order. Field splitting produces two arguments to mv: first a and then b. That is, we run mv a b, overwriting b irrevocably with the contents of a.

My stepper will allow both novices and experienced shell users to observe and understand the shell’s behavior. I conjecture it will be a useful teaching tool. I plan to demonstrate the tool at the workshop and hope to receive feedback.

References


Michael Greenberg. 2017. Understanding the POSIX Shell as a Programming Language. OBT.


