# ithnc is solution is solution in the solution of the solution is solution in the solution is solution in the solution is solution in the solution in the solution in the solution is solution in the solution in the solution in the solution is solution in the solution is solution in the solut

110

# The POSIX shell is an interactive DSL for concurrency

Anonymous Author(s)

### 1 Introduction

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22 23

24 25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

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<sub>1</sub> ... file<sub>n</sub> 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

DSLDI'18, 2018-11-06, SPLASH Boston, MA, USA

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]:

<pre>for k in \${KS}; do</pre>	76
lda est 1/50 \${k} \	77
settings.txt \${ABS} \	78
seeded \${DIR}/lda\${k}	79
done	80
echo "DONE"	81
(a) Sequential	82
(a) Sequentiai	83
<pre>for k in \${KS}; do</pre>	84
lda est 1/50 \${k} \	85
settings.txt \${ABS} \	86
seeded \${DIR}/lda\${k} &	87
done	88
wait	89
echo "DONE"	90
	01

### (b) Concurrent and parallel

In the sequential version (a), we run 1da 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 1da is independent-we can speed things up by building models concurrently. The concurrent version (b) makes only two changes. First, we run 1da 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 1da processes in the same order-but without waiting. 103 We then use the built-in wait utility to have the shell wait 104 for all of its background processes to terminate. In concur-105 rency terminology, we've just used fork/join parallelism: the 106 & is a "fork" and wait is a "join". The speedup from build-107 ing the models in parallel and concurrently on a multicore 108 machine are appreciable-nearly linear. Beyond the speedup, 109

<sup>2018.</sup> ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

185

186

187 188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

Shell expansion step-by-s			
Wondering why the POSIX shell behaves the way it	does? We can expand strings the according to the PC	OSIX step in arbitrary shell and	operating system environm
hell code to expand echo \$(x=6) \$((x+1))		Legend	
ecilo e(x=o) e((x+1))		unexpanded words	initial state
		expanded words	before field splitting
		temporary field	before pathname expansi
		fully expanded field	final state
invironment	Home directories		
("PATH": "/usr/bin", "HOME": "/Users/mgree")	("mgree": "/Users/mgree", "root": "/var/root")		
*		ll.	
Expand shell code	3		
mbolic expansion steps:			
echo \$(x=5) \$((x+1)) Variable	Value		
?	0		
HOME	/Users/mgree /usr/bin		
	74817 0111		
Simple command: begin expansion			
echo \$(x=5) \$((x+1)) Variable	Value		
?	0		
HOME	/Users/mgree		
(a)	Entering a progr	am to step	þ
• 01 (BD10		am to step	þ
?	Entering a progr	am to ster	2
• 01 (BD10	0	am to step	2
? HOME	0 /Users/mgree	am to ster	0
? HOME PATH	0 /Users/mgree /usr/bin 5		)
? HOME PATH X imple commmand: argument expansion step > ArH	0 /Users/mgree /usr/bin 5		)
7 HOME 9ATH × Imple command: argument expansion step > Arit http://witable	0 Alershingree Aursbin 5 hmetic expansion: computed arbitmetic result > 12 Value		)
PHOME PATH ACCOMPANY	0 Aberuhanne Aurzhan 5 hmetic eugansion: conjouted arithmetic result > 10 Value 0		)
7 HOME 9ATH × Imple command: argument expansion step > Arit http://witable	0 Alershingree Aursbin 5 hmetic expansion: computed arbitmetic result > 12 Value		2
7 HOME SMH × x mindle command: argument expanden step > Arill bricks 556 Windde 7 HOME	Ausrohmpree Ausrohm S Meetic expansion computed arithmetic result > 10 Value 0 Ausrohmpree		2
POLICY Comments of the second	Autorshipree Aurobin 5 hmetic expansion computed arithmetic result > 10 0 Autorhin 5		2
P P P P P P P P P P P P P P P P P P P	Autorshipree Aurobin 5 hmetic expansion computed arithmetic result > 10 0 Autorhin 5		2
P HOME PATH A HOME PATH A K  PATH A HOME	0 0 Aben/hgree Aur/bin 5 hmetic expansion conjouted arithmetic result > 10 Value 0 Auer/hgree 4 up/bin 5 4 up/bin 5 4 up/bin 5 4 up/bin 5 4 up/bin 5 4 up/bin 5 4 up/bin 5 4 up/bin 5 4 up/bin 6 up/bin 6 up/bin 7 8 up/bin 7 8 up/bin 8 up/bin 8 up/bin 8 up/bin 8 up/bin 8 up/bin 8 up/bin 8 up/bin 8 up/bin 8 up/bin 9		<u>)</u>
7 1004 1004 1004 1004 1004 1004 1004 100	Ausrihmere Ausrihmere Ausrihmere S Value O Ausrihmere Ausrihmere Ausrihmere Ausrihm S digitting > Gi Pathame expansion Value Q		2
P HOME PATH PATH PATH PATH PATH PATH PATH PATH	0 0 Aben/hgree Aur/bin 5 hmetic expansion conjouted arithmetic result > 10 Value 0 Auer/hgree 4 un/bin 5 4 un/bin 5 4 un/bin 5 4 un/bin 5 4 un/bin 5 4 un/bin 5 4 un/bin 5 4 un/bin 5 4 un/bin 6 un/bin 6 un/bin 7 8 un/bin 7 8 un/bin 8 un/bin 9		)
P P P P P P P P P P P P P P P P P P P	0 Absentingree Aur/bin 5 hinetic expansion computed arithmetic result ≥ § wheetic expansion computed arithmetic result ≥ § 0 Absentingree 5 4 4 4 4 5 5 5 4 4 4 4 4 4 4 4 4 4 4 4 4		2
7 HOAE SATH Kammadia argument expanden stop > Arll Kammadia argument expanden stop > Arll Kammadia argument expanden stop > Arll Kammadia argument expanden stop > Fiel Sath Kammadia argument expande	Ascribin Ascribin Ascribin Ascribin Ascribin Ascribin Ascribin Ascribing S S S S S S S S S S S S S S		)
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2	Ascribin Ascribin Ascribin Ascribin Ascribin Ascribin Ascribin Ascribing S S S S S S S S S S S S S S		)
2 2 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4	a Ausrahmere Au		
7 7 HOASE 7 HOASE 7 HOASE 8 WITH 8 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	Alterningree Alterningree S Value Value 0 Alterningree Autribin 3 statiting > El Pathrume expansion Value 0 Autribin 5 statiting > El Pathrume expansion Value 0 Alterningree Autribin 5 S Wale 0 Alterningree 0 Autribin 0 Alterningree 0 Autribin 5 S S S S S S S S S S S S S		)
2 2 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4	a Ausrahmere Au		

(b) Results of stepping, highlighting field separators

Figure 1. The POSIX shell stepper in action

parallelizing the sequential code is simple-the Levenshtein distance between the two snippets is 7.

### A stepper for the shell 4

143

144

145

146

147

148

149

150

151

152

153

155

159

165

echo 5.6

I have built a web-based stepper that uses a mechanized 154 semantics of the POSIX shell to visualize the various stages of expansion and evaluation (Figure 1). The implementation 156 uses type classes to abstract over operating system features: the stepper uses a symbolic implementation of OS features; a 158 concrete implementation makes real system calls and yields a working shell with identical logic to the symbolic one. 160

Even experienced shell users can be confused by the POSIX 161 shell's semantics, particularly by *field splitting*, a late step in 162 expansion that breaks apart the result of earlier stages into 163 a command's arguments. Mazurak and Zdancewic [2007]'s 164

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 workship and hope to receive feedback.

### References

- David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. J. Mach. Learn. Res. 3 (March 2003), 993-1022. http://dl.acm. org/citation.cfm?id=944919.944937
- Loris D'Antoni, Rishabh Singh, and Michael Vaughn. 2016. NoFAQ: Synthesizing Command Repairs from Examples. CoRR abs/1608.08219 (2016). http://arxiv.org/abs/1608.08219
- Simson Garfinkel, Daniel Weise (Ed.), and Steven Strassman (Ed.). 1994. The UNIX Hater's Handbook. IDG Books Worldwide, Inc., San Mateo, CA, USA.
- Michael Greenberg. 2017. Understanding the POSIX Shell as a Programming Language. OBT.
- Michael Greenberg. 2018. Word expansion supports POSIX shell interactivity. In Programming Companion (presented at Programming eXperience (PX)). ACM. https://doi.org/10.1145/3191697.3214336
- Michael Greenberg, Kathleen Fisher, and David Walker. 2015. Tracking the Flow of Ideas through the Programming Languages Literature. In SNAPL (Leibniz International Proceedings in Informatics (LIPIcs)), Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 140-155. https://doi.org/10.4230/LIPIcs.SNAPL.2015.140
- IEEE and The Open Group. 2016. The Open Group Base Specifications Issue 7 (IEEE Std 1003.1-2008). IEEE and The Open Group.
- Nicolas Jeannerod, Claude Marché, and Ralf Treinen. 2017a. A Formally Verified Interpreter for a Shell-Like Programming Language. In Verified Software. Theories, Tools, and Experiments - 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers. 1-18. https://doi.org/10.1007/978-3-319-72308-2\_1
- Nicolas Jeannerod, Yann Régis-Gianas, and Ralf Treinen. 2017b. Having Fun With 31.521 Shell Scripts. Technical Report hal-01513750.
- Karl Mazurak and Steve Zdancewic. 2007. ABASH: Finding Bugs in Bash Scripts. In PLAS. 105-114. https://doi.org/10.1145/1255329.1255347
- 214 215 216 217

218

219 220

2