Executable Formal Semantics for the POSIX Shell

Smoosh: the Symbolic, Mechanized, Observable, Operational Shell

MICHAEL GREENBERG, Pomona College, USA
AUSTIN J. BLATT*, Puppet Labs, USA

The POSIX shell is a widely deployed, powerful tool for managing computer systems. The shell is the expert's control panel, a necessary tool for configuring, compiling, installing, maintaining, and deploying systems. Even though it is powerful, critical infrastructure, the POSIX shell is maligned and misunderstood. Its power and its subtlety are a dangerous combination.

We define a formal, mechanized, executable small-step semantics for the POSIX shell, which we call Smoosh. We compared Smoosh against seven other shells that aim for some measure of POSIX compliance (`bash`, `dash`, `zsh`, `OSH`, `mksh`, `ksh93`, and `yash`). Using three test suites—the POSIX test suite, the Modernish test suite and shell diagnostic, and a test suite of our own device—we found Smoosh's semantics to be the most conformant to the POSIX standard. Modernish judges Smoosh to have the fewest bugs (just one, from using `dash`'s parser) and no quirks. To show that our semantics is useful beyond yielding a conformant, executable shell, we also implemented a symbolic stepper to illuminate the subtle behavior of the shell.

Smoosh will serve as a foundation for formal study of the POSIX shell, supporting research on and development of new shells, new tooling for shells, and new shell designs.

CCS Concepts: • **Software and its engineering** → **Scripting languages**; **Command and control languages**; *Language features*; *Semantics*; • **General and reference** → *Design*; • **Human-centered computing** → *Command line interfaces.*

Additional Key Words and Phrases: command line interfaces, POSIX, formalization, small-step semantics

ACM Reference Format:
Michael Greenberg and Austin J. Blatt. 2020. Executable Formal Semantics for the POSIX Shell: Smoosh: the Symbolic, Mechanized, Observable, Operational Shell. *Proc. ACM Program. Lang.* 4, POPL, Article 43 (January 2020), 30 pages. https://doi.org/10.1145/3371111

1 INTRODUCTION

The POSIX shell is a line-oriented, potentially interactive scripting language [Austin Group 2018]. The shell is widely used by experts on all three major platforms (Linux, macOS, and Windows); it is often the best—or only!—way to configure, compile, install, manage, deploy, or remove software systems. The shell is used for these tasks because the shell easily combines filesystem manipulations (using standard utilities like `cp`), system management features (like `apt`), and asynchronous job control (using pipelines $c_1 \mid c_2 \mid \ldots$, background jobs c `&`, and the `wait` builtin). Modern container systems rely on the shell: some use the shell extensively (e.g., Docker) while others use the shell as a necessary escape hatch (e.g., Vagrant, Puppet). The POSIX shell is critical software infrastructure.

Critical though it may be, the POSIX shell is not well understood. It has unusual semantics: the general command language doesn't evaluate subexpressions, but rather treats them as strings and *expands* the special control codes in them. This process, called *word expansion*, is responsible for:

*Work done while an undergraduate at Pomona College.

Authors' addresses: Michael Greenberg, Department of Computer Science, Pomona College, Claremont, CA, USA, michael@cs.pomona.edu; Austin J. Blatt, Puppet Labs, Portland, OR, USA, austinblatt@gmail.com.

the translation of ~ into one's home directory; parameter, i.e., variable, substitution; command substitution (written `$(`*c*`)` or `` `c` ``); and globbing with * and ?. Word expansion is simultaneously part of the shell's power and part of its danger [Greenberg 2018b]. It is very easy for word expansion to generate too many or too few arguments to a command... and commands play for keeps! One famous example is "Steam cleaning",[1] though similar bugs abound. The treacherous combination of powerful commands and abstruse semantics has led to no small amount of disgust with the shell (see, for example, the *UNIX Hater's Handbook* [Garfinkel et al. 1994]).

We aim to rehabilitate the shell. The POSIX shell is one of the oldest programming languages still in popular use; it ought to have tool support commensurate with the state of the art. In order to have effective tool support, we need to be able to soundly reason about the shell's semantics. We want tools to summarize scripts' behavior; to calculate scripts' dependencies or preconditions; or to compile scripts to be faster, safer, or in another language.

To that end, we introduce Smoosh: the symbolic, mechanized, observable, operational shell. Smoosh serves as a *mechanized* reference semantics for the POSIX standard. Smoosh is, first and foremost, an *operational* shell: it can be used interactively or for scripting. Smoosh is also *symbolic* and *observable*, generating traces in both real and simulated environments.

Even though it is a completely usable shell, the mechanized small-step operational semantics for Smoosh is relatively compact: the two core stepping functions (Section 5) are a combined total of 1 034 SLOC of Lem. (Lem is an OCaml-like language that can compile to, e.g., OCaml, Coq, etc. [Mulligan et al. 2014].) The small-step nature of the semantics makes it easy to capture and report information. For example, Smoosh can detect and report unspecified and undefined behaviors, trace how and when system calls are made, and log signal handling and traps.

Executability is critical in order to validate our semantics. To show that Smoosh is a good model for the POSIX shell, we compare it against other shells in three different test suites: the POSIX test suite, the Modernish test suite and shell diagnostic [Dekker 2019], and a test suite of our own device (Section 7). We compared Smoosh against seven other shells that aim for some measure of POSIX compliance (`bash`, `dash`, `zsh`, `OSH`, `mksh`, `ksh93`, and `yash`). Smoosh passes all of the locale-independent parts of the POSIX test suite. On all three suites, Smoosh's semantics is the most conformant to the POSIX standard; Modernish judges Smoosh to be the least buggy or quirky. Smoosh's small-step operational semantics uses immutable structures in a functional, GC-ed language, so it is not surprising that Smoosh is slower than the shells written in C on mutable structures: about 4x slower on test suites, but not noticeably slower interactively. On real scripts, we've seen no significant difference in total running time between Smoosh and existing shells. We conjecture that few scripts spend substantial time in shell execution proper compared to programs those shell scripts execute, but it is probable that some real scripts exercise the 4x slowdown. In any case, Smoosh is a *reference* semantics that performs acceptably, not an efficient drop-in replacement for existing shells.

Smoosh isn't just an executable shell, though: Smoosh is parameterized over the operating system state, highlighting the OS interface demanded by a POSIX shell and allowing for alternative uses of the semantics. So far, Smoosh has two implementations of OS interface: a `system` mode where system calls actually occur, and a `symbolic` mode where the system calls are simulated. We use `symbolic` mode to implement a stepper for the shell (Section 6).

We claim the following contributions:

- Smoosh is a new implementation of the POSIX shell specification (Section 2). Its semantics is faithful while still being of manageable size (Section 4).

[1] A misunderstanding of expansion in a Steam script wiped hard drives: http://www.theregister.co.uk/2015/01/17/scary_code_of_the_week_steam_cleans_linux_pcs/.

- We parameterize the Smoosh semantics over a suite of operating system functions (Section 5). The parameter is theoretically useful as it creates a barrier between the shell and the OS; it is practically useful in that it allows for alternative instantiations of OS primitives. As an example, we instantiate the OS with a symbolic mode to write a program stepper, which illuminates the shell's obscure word expansion and evaluation semantics (Section 6).
- Smoosh conforms to the POSIX standard and is suitable as a canonical implementation. Smoosh is the most conformant and least buggy of the seven shells tested (Section 7, Table 1).
- The development of Smoosh led to the identification of numerous bugs in shells in active use (`bash`, `dash`, `yash`, `OSH`) as well as in the POSIX specification and its test suite (Section 7.1).

These contributions lay the foundation for serious tools for improving the shell. Without a semantics, what could we prove about an analysis or compiler for the shell? Smoosh's semantics can be a reference implementation of POSIX for desugarers (like CoLiS [Jeannerod 2017]) and compilers to test and prove against, as well as a semantics against which to prove static analyses sound. Our tested semantics will be a baseline for work improving on the shell's implementation and design.

2 WHAT IS THE POSIX SHELL?

There are two perspectives on what defines the POSIX shell: one places the standard first, and the other places shell implementations first. We aim for Smoosh to accommodate both perspectives.

The first, bureaucratic perspective centers the POSIX specification ([Austin Group 2018], Volume "Shell & Utilities"), which comprises 21 pages of introduction (§1), 98 pages of core definitions (§2 "Shell Command Language"), and additional documentation on 160 utilities (ranging in scale from `awk` to `true`).[2] The part most relevant to shell implementors and programmers is Volume "Shell & Utilities" §2 "Shell Command Language", which weaves together explanation of the POSIX shell's lexer, parser, and semantics. Some important information is left to the description of the `sh` command in §4. The specification is written in the typical style, with specific definitions of words like 'can', 'may', and that troublesome pair, 'unspecified' ("a value or behavior not specified by POSIX.1-2017 which results from use of a valid program construct or valid data input") and 'undefined' ("…which results from use of an invalid program construct or invalid data input"). There are 70 uses of 'unspecified' and 17 uses of 'undefined' relevant to the shell.

The second, pragmatic perspective is that the POSIX shell is a collection of language implementations (e.g., `bash`, `dash`, `yash`) that more or less agree on a core set of features; the POSIX specification is a document that tries to track what that core set is, knowing that few implementations will conform in all respects.

We consider both of these points of view when we discuss conformance (Section 7). When we refer to shell implementations, we mean the versions in Debian 9 (stretch): `bash` 4.4-12(1), `dash` 0.5.6-2.4, `zsh` 5.3.1-4+b2, `OSH` 0.6.pre21, `mksh` 54-2+b4, `ksh93` 20120801-3.1, and `yash` 2.43-1.

There is already precedent for taking something like the POSIX specification and producing a formal model, with recent work on C being a good example [Blazy and Leroy 2009; Ellison and Rosu 2012; Kang et al. 2015; Krebbers et al. 2014; Memarian et al. 2016]. There is similar precedent for taking a collection of implementations and producing a formal model, with λ_{JS} being a good example [Guha et al. 2010]. What is special about doing this process for the POSIX shell? The shell has two distinctive features that make its semantics more interesting:

- **Word expansion.** While typical languages evaluate an expression by evaluating its subexpressions, many shell expressions evaluate their parts by performing *word expansion* rather than recursive evaluation. Word expansion and evaluation are mutually recursive via *command substitutions* `$(`c`)`, which evaluate the command c during expansion. (See Section 2.1.)

[2] We write out 'Section' when referring to sections in this paper and use § to refer to sections of the POSIX specification.

```
$ echo ~root
/var/root
$ usr=root
$ echo ~$usr
~root
```

(a) Tilde expansion

```
$ x=$(ls)
$ echo $x,${#x},${x#*[ab]},${x##*[ab]}.
a b c,5, b c, c.
```

(b) Command substitution, parameter expansion

```
$ y=42 x=5
$ echo $((y += $x))
47
$ echo $((y)) $y
47 47
```

(c) Arithmetic expansion

```
$ ls
a b c
$ x="a b"
$ ls $x
a b
$ ls "$x"
ls: a b: No such file or directory
```

(d) Field splitting

```
$ echo a*
ap app appall apparition appendix applejack
$ echo ap?
app
$ echo appa*
appall apparition
$ echo ap[=p=]*a*
appall apparition applejack
$ echo "a*"
a*
```

(e) Pathname expansion (a/k/a globbing), quote removal

Fig. 1. Expansion examples

- **System calls.** While typical languages make system calls via library functions, the shell makes system calls in its semantics. Core operations depend on forking a new process (`fork`), replacing the current process with an executable (`execve`), waiting on processes (`wait` and `waitpid`), handling and sending signals (`signal`, `kill`), working with files (`open`, `close`, `access`, `stat`, `lstat`, `write`, `read`), and creating and manipulating file descriptors (`pipe`, `fcntl`, `dup2`).

These two features present different problems when trying to understand the shell well enough to construct a sound, working model. Modeling expansion is tricky because it is (a) mutually recursive with evaluation; (b) a subtly structured, four-stage process; and (c) used by the evaluation semantics in several slightly different configurations. Modeling system calls is tricky because (a) there are many of them, (b) their behavior is subtle, and (c) giving faithful meaning to them can bring arbitrarily much of the operating system into the formal model.

2.1 What is word expansion?

The POSIX specification indicates that there are four stages of expansion with a total of seven components: (1) tilde expansion, parameter expansion, command substitution, and arithmetic expansion; (2) field splitting; (3) pathname expansion; and (4) quote removal. Shells perform this word expansion left-to-right in stage order (see Section 4.2 for Smoosh's semantics). These seven components are best illustrated by brief examples (Figure 1). The examples in Figures 1b and 1d take place in a directory with three files (`a`, `b`, and `c`); the example in Figure 1e takes place in a different directory, where the result of the first `echo` indicates which files begin with the letter `a`. For a more detailed description of these phases, we refer readers to a longer explanation in our prior work [Greenberg 2018b] and to the POSIX specification [Austin Group 2018] §2.6.

Commands	c	::=	$(s{=}w)^*\ w\ r^*$ | $\mathsf{pipe}_!\ c^+\ \&^?$ | $c\ r^+$ | $c\ \&$ | $(\ c\ )$ | $c_1\,;c_2$ | $c_1 \&\& c_2$ | $c_1 \mid\mid c_2$ | $!c$ | $\mathtt{while}\ c_1\ c_2$ | $\mathtt{for}\ s\ w\ c$ | $\mathtt{if}\ c_1\ c_2\ c_3$ | $\mathtt{case}\ w\ cb^*$ | $s()\ c$
Redirections	r	::=	$\mathtt{file}\ \mathtt{fd}\ ft\ w$ | $\mathtt{dup}\ \mathtt{fd}\ dt\ w$ | $\mathtt{here}\ \mathtt{fd}\ ht\ w$
File redirections	ft	::=	$>$ | $>\mid$ | $<$ | $<>$ | $>>$
Dup redirections	dt	::=	$>\&$ | $<\&$
Heredoc redirections	ht	::=	$\mathtt{default}$ | $\mathtt{noexpand}$
Case branches	cb	::=	$(w^+)\ c$
Words	w	::=	$(s$ | ␣ | $k)^*$
Control codes	k	::=	$\sim s^?$ | $\$\{s\ \phi\}$ | $\$(c)$ | $\$((w))$ | $"w"$ |
Parameter formats	ϕ	::=	normal | $\mathsf{default}_{null-}\ w$ | $\mathsf{assign}_{null=}\ w$ | $\mathsf{error}_{null?}\ w$ | $\mathsf{alt}_{null+}\ w$ | $\mathsf{length}_{\#}$ | $\mathsf{sub}\ side\ mode\ w$
Treatment of null	$null$	::=	string | $\mathsf{unset}_:$
Substring side	$side$	::=	$\mathsf{prefix}_{\#}$ | $\mathsf{suffix}_{\%}$
Substring mode	$mode$	::=	$\mathsf{shortest}_{\mathsf{s}}$ | $\mathsf{longest}_{\mathsf{ss}}$
Non-empty strings	s	$\in$	Σ^+ (e.g., UTF-8)
File descriptors	$\mathtt{fd}$	$\in$	$\mathbb{N}$

Fig. 2. The shell's source syntax

2.2 Smoosh: a foundational, formal interpretation of the POSIX specification

Our work aims to offer a formal interpretation of the specification that is also a conforming implementation. Such a compromise is imperfect: we don't model all possible behaviors in the specification, but rather *interpret* the specification deterministically. Our formal model is in code and not a tiny core calculus. Nevertheless, we believe our model is interesting and our implementation is useful. Analogously, the CompCert compiler doesn't conform to MSVC, ICC, GCC, or Clang exactly, and yet its model of the C language can be seen as canonical [Blazy and Leroy 2009; Krebbers et al. 2014]. Building such a foundational model is a necessary first step towards smaller calculi and more useful tools in a setting where implementations (currently and unyieldingly) disagree on a number of issues (see Section 8).

Our paper has three parts: a formal description, to show that while the shell is complex, it is amenable to conventional techniques (Sections 3 and 4); a description of our implementation, to give more concrete detail on our semantics (Sections 5 and 6); and a discussion of conformance, showing that we have adequately modeled the POSIX shell (Section 7) and extensions (Section 8).

3 SYNTAX

The shell has an idiosyncratic concrete syntax, which we mirror in our abstract syntax (Figure 2) but do not follow strictly. Since we're giving abstract syntax, we ignore the details resolved during parsing, particularly backslash escapes of reserved symbols and single quoting. A few conventions: we use `fixed-width` fonts for system concepts (e.g., `fd`) and for abstract syntax that mirrors the shell's concrete syntax (e.g., `>|`). We use sans-serif fonts for Smoosh concepts and *italics* for metavariables. We occasionally use subscript "hints" to translate English names into concrete shell syntax (e.g., $\mathsf{prefix}_{\#}$). We write $x^?$ for optional nonterminals. We use Kleene star x^* to represent lists and x^+ to represent non-empty lists. We use n to refer to natural numbers and b to refer to booleans, tagging such variables with a descriptive name. For example, $n_{\$?}$ is a number representing an exit status and $b_{\$()}$ is a boolean that indicates whether a command substitution has been performed (which helps determine the exit status; see CmdAssignDoneNoCmd in Figure 11).

The two primary AST constructs are the *command*, c, and *words*, w. Commands are the top-level construct: a user enters a command c which has words w; the words are *expanded* and the command is eventually *evaluated*.

The base case for the command AST is the *simple command*. Simple commands $(s{=}w)^*\ w\ r^*$ model command invocations (i.e., builtins, functions, and executables), plain assignments, and plain redirections. All other command forms are composite. First, there are some "modifiers" of simple commands: pipelines, $\texttt{pipe}_{|}\ c^+\ \&^?$, which may be in the background; redirected commands, $c\ r^+$; background commands a/k/a asynchronous commands, c &; and subshells, (c), noting that curly braces are used for disambiguating parsing. There are sequencing commands and logic: sequence, $c_1\,;c_2$; short-circuiting conjunction, $c_1 \&\& c_2$; short-circuiting disjunction, $c_1 \,||\, c_2$; and negation, $!c$. There are two iteration constructs—while loops, `while` $c_1\ c_2$; and for loops, `for` $s\ w\ c$—and two conditional constructs—the conventional conditional, `if` $c_1\ c_2\ c_3$; and string pattern matching, `case` $w\ cb^*$, where each *case branch* cb pairs one or more patterns (as words w) with a command c to run when a pattern matches. Function definition, s`()` c, is also a command.

Redirections r come in three fundamental forms: to a file (file), to an existing file descriptor (dup), and from a given string a/k/a heredoc (here). File redirections, file `fd` $ft\ w$, specify a source file descriptor, a file mode ft (e.g., > to write to a file), and a file target w. File descriptor redirections, `dup fd` $dt\ w$, can copy and close file descriptors. Finally, heredoc redirections, `here fd` $ht\ w$, expand words w and make the resulting string available for reading on a given file descriptor `fd`. Redirections are typically scoped—that is, they take effect only for a given simple command or composite redirection command; the `exec` special builtin suppresses scoping, causing the redirection to take permanent, global effect in the shell (see discussion of `runCmd` in Section 4.3).

Words w are the primary sub-part of commands. Words are subject to expansion, showing up in several places: in the assignments and arguments of simple commands; as the targets of redirections; as the strings iterated over in `for` loops; and as both the scrutinees and patterns of `case` conditionals. Words w are possibly empty lists of one of: a *user string*—a non-empty string s from some character set, e.g., UTF-8, which we designate Σ; a field separator ␣ (i.e., whitespace, which is statically parsed); or a *control code*, k. Our AST only models lists of words, not individual words, leading to some mismatches with the POSIX specification. For example, shell syntax only allows a single word in the assignment part of simple commands, but our AST allows multiple words to appear in an assignment. Our choice of word representation is motivated by three things: a desire for uniformity (most expansions use lists of words rather than single words), correct handling of empty fields (formed by two adjacent field separators), and our interface with `dash`'s parser (see Section 5.1). `dash`'s parser should never produce an assignment with multiple words.

In our abstract syntax, escaped control codes—like the literal `\$`—are ordinary strings. In the initial stages of word expansion, only control codes are expanded. One might expect `*` to be a control code, but pathname expansion is a dynamic search on the results of expansion and is not statically parsed.

There are five control codes: tilde-and-prefix, $\texttt{\textasciitilde}s^?$, is used to refer to users' home directories, noting that the tilde prefix $s^?$ may be empty and that the so-called 'prefix' in fact comes after the tilde; parameters `${`$s\ \ \phi$`}` take a variable name s and a *parameter format* ϕ which will determine how the results of looking up s will be used; command substitutions `$(`c`)` hold commands that ought to be run with their output captured; arithmetic substitutions `$((`w`))` hold words w that will be further expanded and then parsed as an integer arithmetic expression; and quotation `"`w`"` inhibits field splitting and pathname expansion.

Parameter formats come in two flavors: they allow for convenient defaulting when variables are unset or null (i.e., hold the empty string); and they enable post-processing the result of variable lookup. The normal format is the default and performs standard variable lookup. There are four

defaulting parameter formats: default, assign, error, and alt. All four take words w and a flag *null*: when *null* = string, then a variable set to the empty string is still considered set; when *null* = unset$_:$ (where : is a concrete syntax hint), then a variable set to the empty string is considered unset. When default is used on a variable considered unset (because it's actually unset or because *null* = unset$_:$), then the words w are returned for further expansion (rather than the empty string). When assign is used on a variable considered unset, the words w are returned for further expansion—and the result of expanding those words is assigned to s. When error is used on a variable considered unset, the words w are returned for further expansion—and then used as an error message. The alt format is the *opposite* of default: when it is used on a variable considered unset, the null string is returned; when it is used on a variable considered set, the words w are expanded and returned. The two post-processing parameter formats are length$_\#$ and sub *side mode* w. The former calculates the length of the value of the given parameter; the latter does prefix$_\#$ or suffix$_\%$ removal using either a shortest$_s$ or longest$_{ss}$ match policy on a given pattern w.

We omit two shell features from our abstract syntax: our parser desugars `until` loops into `while` loops, and the tab-stripping heredoc redirection `<<-` is handled in the parser. Other bits of concrete syntax are ignored: infix and postfix keywords for conditionals and loops (e.g., `then`, `esac`); our AST doesn't allow you to omit the first parenthesis in a case branch; quoted heredocs (`<<"EOF"`) are marked noexpand, as they will not undergo expansion.

4 SEMANTICS

POSIX specifies a broad set of behaviors for the shell; our semantics is not small. We show excerpts here of the shell's semantics to (a) show that the shell is nevertheless amenable to standard techniques, and (b) give a sense of the level of detail of our semantics. This section is a selective transcription of Smoosh's implementation (Section 5) into inference rules, meant to communicate the core ideas in a concise mathematical form without minutae (like logging or symbolic value manipulation). We only ever show excerpts of our actual rules—the 'real' rules are in our code. We *do* show all of our state definitions (Section 4.1, Figure 3) and all of the intermediate forms used by our small step semantics (Figure 4). We take care to lay out these forms of state plainly, even if we do not give our entire semantics in mathematical notation.

We explain expansion (Section 4.2) before evaluation (Section 4.3). Rather than give an up-front description of every helper function and system call, we introduce them as needed. We write the types of these functions in 'schematic form', using metavariables rather than named types.

4.1 State

The POSIX shell has to track a significant amount of state (Figure 3). Each line of the description of Smoosh's shell state characterizes different levels of detail. Before explaining those lines in detail, we give a modest overview of shell concepts. The POSIX standard specifies the minimum state for a shell in §2.12. Confusingly, some of the specified state is kept by the OS, not the shell (e.g., "open files inherited upon invocation of the shell, plus open files controlled by exec" and "file creation mask set by `umask`"). The shell executable starts as an *outermost* shell (our terminology). *Subshells* are forked copies of the shell running on some other command. Quoting from §2.12, subshells are a "duplicate of the shell environment, except that signal traps that are not being ignored shall be set to the default action". Subshells are used for "command substitution, commands that are grouped with parentheses, and asynchronous lists" (i.e., background commands). Any "[c]hanges made to the subshell environment shall not affect the shell environment". The shell is dynamically scoped, and subshells inherit their parent shell's environments and settings. No information flows from subshells to their parents other than the subshells exit status, any output captured by the parent shell, and global system effects (e.g., a subshell can send signals to its parent).

Shell state	σ	::=	$\langle \mathtt{pid}_{\mathsf{root}}, b_{\mathsf{outermost}}, \mathsf{opts}, \mathsf{jobs}, \mathsf{traps}, \mathsf{traps}^{?}_{\mathsf{supershell}},$ $\rho, s^{*}_{\$*}, \ell^{*}, \mathcal{V}_{\mathsf{ro}}, \mathcal{V}_{\mathsf{export}}, \rho_f, \mathsf{aliases}, \mathsf{locale},$ $s_{\mathsf{cwd}}, \mathtt{pid}_{\$!}, n_{\$?}, n_{\mathsf{loop}}, n^{?}_{\mathsf{optoff}} \rangle$
Shell options	opts	$\in$	$\mathsf{Opts} = \{\mathtt{allexport}, \dots\}$
Traps	traps	:	$\mathsf{Sig} \rightharpoonup s$
Jobs	jobs	:	$id{:}\mathbb{N} \rightharpoonup \{\mathsf{ji} \mid \mathsf{ji}.n_{\mathsf{id}} = id\}$
Job info	ji	::=	$\langle n_{\mathsf{id}}, (\mathtt{pid}\ c)^{*}_{\mathsf{pipe}}, \mathtt{pid}, c, js \rangle$
Job status	js	::=	running | stopped (TSTP | STOP | TTIN | TTOU) | terminated sig | done $n_{\$?}$
Signals	sig	$\in$	$\mathsf{Sig} = \{\mathsf{SIGHUP}, \dots\}$
Global environments	ρ	:	$s \rightharpoonup s$
Local environments	ℓ	:	$s \rightharpoonup s^{?} \times b_{\mathsf{ro}} \times b_{\mathsf{export}}$
Sets of variable names	$\mathcal{V}$	$\subseteq$	$\mathcal{P}(\Sigma^{+})$
Function definitions	ρ_f	:	$s \rightharpoonup c$
Aliases	aliases	:	$s \rightharpoonup s$
Locales	locale	$\in$	$\mathcal{L}$ (e.g., C, it_IT.UTF-8)

Fig. 3. The shell's state

The first line is about high-level process info. The root process ID, $\mathtt{pid}_{\mathsf{root}}$, is used for the special parameter $$ (even in subshells). Each shell knows whether it is the outermost shell or a subshell ($b_{\mathsf{outermost}}$), since only the outermost shell should perform certain job control and signal-handling functions. Shell options are tracked in opts; current jobs are tracked in (jobs). Smoosh tracks not only the current shell's signal handlers a/k/a traps (traps), but also any traps from the supershell ($\mathsf{traps}^{?}_{\mathsf{supershell}}$). In order to find out which traps are currently set without writing to a file, but `$(traps --)` will execute in a subshell. By tracking the supershell's traps, we can implement the optional POSIX behavior of subshells inheriting the supershell's traps for display purposes. The traps field is a partial function from signals to strings. Signals not in the domain of $\sigma.\mathsf{traps}$ have default dispositions; signals that map to empty strings are ignored; otherwise, the strings are parsed and interpreted when signals are handled (see checkTraps in Figure 10).

The second line of the shell's state characterizes the environment: ρ is the global, dynamically scoped environment; the current positional parameters are stored in a list $s^{*}_{\$*}$. In addition to environment and positional parameters, Smoosh also tracks a stack of local environments, ℓ^{*}. These local environments exist not only to support the non-POSIX builtin `local`, but also for scoped assignments for function calls (see Section 8.3). The shell tracks the read-only and exported variables in ρ via $\mathcal{V}_{\mathsf{ro}}$ and $\mathcal{V}_{\mathsf{export}}$, respectively. Aliases and the current locale information are also tracked. (Smoosh only supports the ambient locale, though; see Section 5.1.)

The third and final line of the shell's state holds finer grained information: the current working directory (s_{cwd}), the PID of the last background command ($\mathtt{pid}_{\$!}$), the exit status of the last command ($n_{\$?}$), how deeply nested we are in the current loop (n_{loop}), and the offset into the argument for parsing in `getopts` ($n^{?}_{\mathsf{optoff}}$) (see Section 8.2).

Smoosh's state is just one way to keep track of everything a shell needs to know. For example, the entire last line of the shell state could instead be kept in the shell's environment, $\sigma.\rho$. Some shell state in fact *must* be kept there, e.g., the `OPTIND` variable used by the `getopts` builtin. We find it more convenient to manually track the last exit status in $\sigma.n_{\$?}$ as a number than to repeatedly coerce a string-valued numerical representation from an environment variable in $\sigma.\rho$.

$$
\begin{array}{lrcl}
\text{Commands} & c & ::= & \ldots \mid \mathsf{cmd}_{\mathsf{args}}\ (s{=}w)^*\ es\ r^*\ co \mid \mathsf{cmd}_{\mathsf{redirs}}\ (s{=}w)^*\ f\ rs\ co \mid \\
& & & \mathsf{cmd}_{\mathsf{assigns}}\ (s{=}es)^*\ f\ sfds\ co \mathsf{cmd}_{\mathsf{ready}}\ \rho\ s_{\mathsf{cmd}}\ f_{\mathsf{args}}\ sfds\ co \mid \mathsf{run}\ \rho_{\mathsf{cmd}}\ s_{\mathsf{cmd}}\ f_{\mathsf{args}}\ sfds\ co \mid \\
& & & \mathsf{while}_{\mathsf{cond}}\ c_{\mathsf{orig}}\ c_{\mathsf{cur}}\ c_{\mathsf{body}} \mid \mathsf{while}_{\mathsf{body}}\ c_{\mathsf{cond}}\ c_{\mathsf{body}}\ c_{\mathsf{cur}} \mid \\
& & & \mathsf{for}_{\mathsf{args}}\ s\ es\ c \mid \mathsf{for}_{\mathsf{start}}\ s\ f\ c \mid \mathsf{for}_{\mathsf{running}}\ s\ f\ c_{\mathsf{body}}\ c_{\mathsf{cur}} \mid \\
& & & \mathsf{case}_{\mathsf{arg}}\ es\ cb^* \mid \mathsf{case}_{\mathsf{match}}\ s\ cb^* \mid \mathsf{case}_{\mathsf{check}}\ s\ es_{\mathsf{pat}}\ c\ cb^* \mid \\
& & & \mathsf{call}\ n_{\mathsf{loop}}\ s^*_{\$*}\ s_{\mathsf{fun}}\ c_{\mathsf{orig}}\ c_{\mathsf{cur}} \mid \mathsf{break}\ n \mid \mathsf{continue}\ n \mid \mathsf{return} \mid \mathsf{exit} \mid \mathsf{done} \mid \mathsf{redirs}\ c\ sfds \mid \\
& & & \mathsf{eval}\ n_{\mathsf{linno}}\ \mathtt{pctx}\ s_{\mathsf{src}}\ b_{\mathsf{interactive}}\ b_{\mathsf{toplevel}} \mid \mathsf{eval}_{\mathsf{cmd}}\ n_{\mathsf{linno}}\ \mathtt{pctx}\ s_{\mathsf{src}}\ b_{\mathsf{interactive}}\ b_{\mathsf{toplevel}}\ c \mid \\
& & & \mathsf{exec}\ s_{\mathsf{path}}\ s_{\mathsf{name}}\ f_{\mathsf{args}}\ \rho\ b_{/\mathsf{bin/sh}} \mid \mathsf{wait}\ \mathtt{pid}\ b_{\mathsf{checked}}\ b_{\mathsf{cmd}} \mid \\
& & & \mathsf{checked}\ c \mid \mathsf{trapped}\ \mathtt{sig}\ n_{\$?}\ c_{\mathsf{handler}}\ c_{\mathsf{cont}} \\
\text{Command options} & co & ::= & \langle b_{\$()}, b_{\mathsf{fork}}, b_{\mathsf{simple}} \rangle \\
\text{Redirection state} & rs & ::= & \langle er^*\ xr^?\ r^* \rangle \\
\text{Expanded redir} & er & ::= & \mathsf{efile}\ ft\ \mathtt{fd}\ s \mid \mathsf{edup}\ dt\ \mathtt{fd}\ \mathtt{fd}^? \mid \mathsf{ehere}\ ht\ \mathtt{fd}\ s \\
\text{Expanding redir} & xr & ::= & \mathsf{xfile}\ ft\ \mathtt{fd}\ es \mid \mathsf{xdup}\ ft\ \mathtt{fd}\ es \mid \mathsf{xhere}\ ht\ \mathtt{fd}\ es \\
\text{Control codes} & k & ::= & \ldots \mid \mathsf{assign}\ s\ w \mid \mathsf{error}\ s\ w \mid \mathsf{match}\ f\ side\ mode\ w \mid \\
& & & \mathsf{cmdsubst}\ c\ \mathtt{pid}\ \mathtt{fd} \mid \mathsf{cmdwait}\ c\ \mathtt{pid}\ s \\
\text{Expansion state} & es & ::= & \mathsf{start}\ eo\ w \mid \mathsf{expand}\ eo\ e\ w \mid \mathsf{split}\ eo\ e \mid \\
& & & \mathsf{path}\ eo\ i \mid \mathsf{quote}\ eo\ i \mid \mathsf{error}\ f \mid \mathsf{done}\ f \\
\text{Expansion options} & eo & ::= & \langle b_{\mathsf{split}}, b_{\mathsf{glob}} \rangle \\
\text{Expanded words} & e & ::= & (\text{␣} \mid \mathsf{src}\ s \mid \mathsf{exp}\ s \mid \mathtt{@}\ f \mid \text{"}s\text{"})^* \\
\text{Intermediate fields} & i & ::= & (\mathsf{ws}\ \text{␣} \mid \text{␣} \mid s \mid \text{"}s\text{"})^* \\
\text{Fields} & f & ::= & s^* \\
\text{Parsing context} & \mathtt{pctx} & \in & \mathcal{L}\ \text{(parser state)} \\
\text{Saved FDs} & sfds & : & \mathtt{fd} \rightharpoonup ofd \\
\text{Old FD info} & ofd & ::= & \mathsf{restore}\ \mathtt{fd} \mid \mathsf{close}
\end{array}
$$

Fig. 4. Smoosh's runtime extensions for use in the small-step semantics

Taking Smoosh's general approach as a given, there are still alternative designs. For example, we track an explicit stack of local variables in the shell state σ, but we don't have such a stack for function parameters: instead we use call stack frames to save the old positional parameters (Figure 4). Not having a stack of positional parameters makes it easy to ensure that we never have an underflow of the positional parameter stack; having an explicit stack of locals makes it easier to do scoped lookup. Our variable lookup routine works as follows: it immediately returns the value for a special variable (e.g., the value of `$?` is stored in $\sigma.n_{\$?}$) or positional parameter (in $\sigma.s^*_{\$*}$); failing that, it traverses the stack of locals $\sigma.\ell^*$; failing that, it checks the global environment $\sigma.\rho$.

4.2 Word expansion

Word expansion occurs when evaluating assignments, commands, redirections, the iteratee of a `for` loop, and both the scrutinee and the patterns of `case` conditionals. The general outline of the process takes *words* w and expands them to a list of strings, called *fields* f. We sketched the features of word expansion by example already (Section 2.1); we now give a formal model.

We model expansion as a transition system between expansion states *es* (Figure 4). There are seven such expansion states (graphically in Figure 5; formally in Figure 6), not to be confused with POSIX's seven components of its four stages. The states are: the initial state, start *eo* w; initial word expansion, expand *eo* e w, where we perform the first stage of POSIX word expansion (tildes, parameters, command substitution, and arithmetic); field splitting, split *eo* e; pathname expansion,

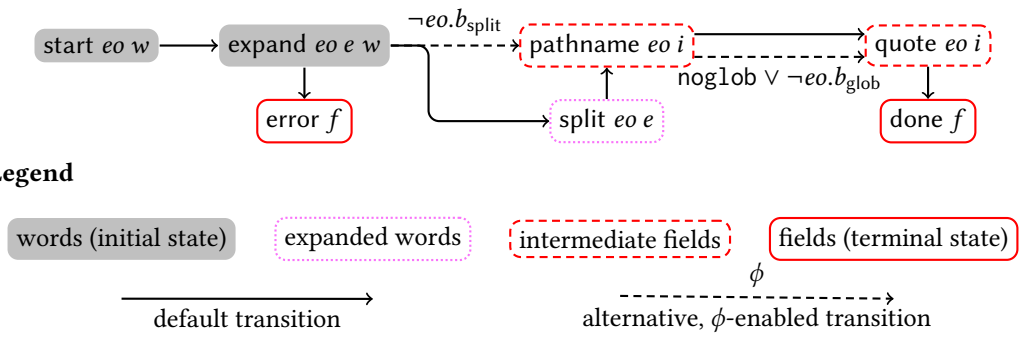

Fig. 5. Word expansion transitions

path *eo i*; quote removal, quote *eo i*; and two terminal states, one for errors, error f, and one for successful completion, done f.

A diagram of the possible transitions (Figure 5) uses boxes to indicate the different representations used for words during expansion (gray for words w, the initial state; dotted pink lines for expanded words, e; dashed red lines for intermediate fields, i; and solid red lines for fields, f, the final result). Solid transitions are defaults and dashed transitions are alternatives enabled under the predicated conditions. The transition graph shows: that errors can only occur during control code expansion; that field splitting can be skipped wholesale (dashed line); and that pathname expansion is always run, but sometimes it doesn't actually glob but rather unescapes globbing characters (dashed line). We give a lower level understanding of the possible transitions (Figure 6) with inference rules for a small-step semantics, where $\sigma, es \longrightarrow \sigma', b_{\$()}, es$ means that in shell state σ, the expansion state es transitions to a new shell state σ' and a new expansion state es. The boolean flag $b_{\$()}$ indicates whether or not a command substitution was run. The transition system conditions on the *expansion options*, *eo*, which are held in es.

The shell has multiple notions of errors, which lead to different behavior in different circumstances (see §2.8.1 "Consequences of Shell Errors"). Like JavaScript, many erroneous-seeming conditions are silently ignored, e.g., if a glob doesn't match any files, then it is treated literally. Non-interactive shells will exit on most errors, while interactive ones will continue unless `errexit` is set.

Initial word expansion in the expansion state expand is defined as a small-step operational semantics $\sigma, wo, e, w \longrightarrow \sigma, b_{\$()}, wer$, where the initial e and w represent the current progress expanding the words w into the *expanded words* e (or an error, as recorded in the *intial word expansion result* wer). We track whether or not a command substitution is performed ($b_{\$()}$), along with a variety of options particular to initial word expansion (wo): whether splitting should happen (for a special case of `$*`), whether the current control codes are quoted, and whether the current control codes are from user input or are indirectly generated.

The rules map over each component of the words under expansion, pulling an element off the front, processing it, and adding the result to the already expanded words e. We offer here only an excerpt of the control code expansion rules: those dealing with command substitution (Figure 7). Command substitution is exemplary because (a) it ties the recursive knot between evaluation and expansion, (b) it makes several system calls, and (c) its evaluation produces more control codes.

Given a command substitution control code `$(`c`)`, we must run the command c, capturing its output and saving it for further expansion. The CmdSubst rule starts the process. First, we use the `pipe` system call to create a POSIX pipe, an OS-managed FIFO queue with a file descriptor at each end: one for reading ($\texttt{fd}_r$) and one for writing ($\texttt{fd}_w$). Next, we fork a subshell, running our

Word expansion $\boxed{\sigma, es \longrightarrow \sigma, b_{\$()}, es}$

$$\frac{}{\sigma, \text{start } eo\ w \longrightarrow \sigma, \bot_{\$()}, \text{expand } eo\ \cdot\ w} \quad \textsc{ExpStart}$$

$$\frac{\sigma, \langle eo.b_{\text{split}}, \bot^{\text{"}}, \bot_{\text{gen}}\rangle, e, w \longrightarrow \sigma, b_{\$()}, \langle e', w'\rangle}{\sigma, \text{expand } eo\ e\ w \longrightarrow \sigma', b_{\$()}, \text{expand } eo\ e'\ w'} \quad \textsc{ExpExpand}$$

$$\frac{\sigma, \langle eo.b_{\text{split}}, \bot^{\text{"}}, \bot_{\text{gen}}\rangle, e, w \longrightarrow \sigma, b_{\$()}, \text{error } e_{\text{err}}}{\sigma, \text{expand } eo\ e\ w \longrightarrow \sigma', b_{\$()}, \text{error toFields}(e_{\text{err}})} \quad \textsc{ExpExpandErr}$$

$$\frac{eo.b_{\text{split}}}{\sigma, \text{expand } eo\ e\ \cdot \longrightarrow \sigma, \bot_{\$()}, \text{split } eo\ e} \quad \textsc{ExpExpandSplit}$$

$$\frac{\neg eo.b_{\text{split}}}{\sigma, \text{expand } eo\ e\ \cdot \longrightarrow \sigma, \bot_{\$()}, \text{path } eo\ \text{skipSplitting}(e)} \quad \textsc{ExpExpandNoSplit}$$

$$\frac{}{\sigma, \text{split } eo\ e \longrightarrow \sigma, \bot_{\$()}, \text{path } eo\ \text{fieldSplitting}(\sigma, e)} \quad \textsc{ExpSplit}$$

$$\frac{\texttt{noglob} \notin \sigma.\text{opts} \qquad eo.b_{\text{glob}}}{\sigma, \text{path } eo\ i \longrightarrow \sigma, \bot_{\$()}, \text{quote } eo\ \text{pathnameExpansion}(\sigma, i)} \quad \textsc{ExpPath}$$

$$\frac{\texttt{noglob} \in \sigma.\text{opts} \vee \neg eo.b_{\text{glob}}}{\sigma, \text{path } eo\ i \longrightarrow \sigma, \bot_{\$()}, \text{quote } eo\ \text{unescape}(i)} \quad \textsc{ExpPathNoGlob}$$

$$\frac{}{\sigma, \text{quote } eo\ i \longrightarrow_{\bot} \sigma, \bot_{\$()}, \text{done combineFields}(\text{removeQuotes}(i))} \quad \textsc{ExpQuote}$$

Helper function	Description
$\text{skipSplitting}(e) = i$	Convert expanded words to intermediate fields
$\text{toFields}(e) = f$	Convert expanded words to fields
$\text{fieldSplitting}(\sigma, e) = i$	Break expanded words into intermediate fields
$\text{unescape}(i) = i$	Remove glob escape characters
$\text{pathnameExpansion}(\sigma, i) = i$	Expand globs (e.g., * and ?)
$\text{removeQuotes}(i) = i$	Remove quote characters/AST nodes
$\text{combineFields}(i) = f$	Conjoin adjacent fields and whitespace

Fig. 6. Small-step semantics for word expansion

command c with some redirections: first, standard output is redirected to the write end of our pipe, $\texttt{fd}_w$; second, inside of the forked process, we close the read end of our pipe, $\texttt{fd}_r$. Finally, we close the write end of the pipe in our running shell, stepping to an intermediate cmdsubst form that tracks both the `pid` of the forked subshell and the pipe's read end. With the command running, we try to read all of the file descriptor (using the system call `readAll`; CmdSubstRead). Once we've read everything, we close $\texttt{fd}_r$, save the string, and use the cmdwait form to wait on the command to finish. If reading produces an error, the entire expansion process errors out (CmdSubstReadErr).

Initial word expansion $\boxed{\sigma, wo, e, w \longrightarrow \sigma, b_{\$()}, wer}$

Initial word expansion options $wo ::= \langle b_{\text{split}}, b^{\text{»}}, b_{\text{gen}} \rangle$
Initial word expansion results $wer ::= \langle e, w \rangle \mid \text{error } e$

$$lit = \begin{cases} \text{"}s\text{"} & wo.b^{\text{»}} \\ \text{exp } s & \neg wo.b^{\text{»}} \wedge wo.b_{\text{gen}} \\ \text{src } s & \neg wo.b^{\text{»}} \wedge \neg wo.b_{\text{gen}} \end{cases}$$

$$\frac{}{\sigma, wo, e, \llcorner\ w \longrightarrow \sigma, \bot_{\$()}, \langle e\ \llcorner, w \rangle}\ \textsc{EWSep} \qquad \frac{}{\sigma, wo, e, s\ w \longrightarrow \sigma, \bot_{\$()}, \langle e\ lit, w \rangle}\ \textsc{EWLit}$$

$$\frac{\sigma, wo, k \longrightarrow \sigma', b_{\$()}, \langle e', w' \rangle}{\sigma, wo, e, k\ w \longrightarrow \sigma, b_{\$()}, \langle e\ e', w'\ w \rangle}\ \textsc{EWCtrl} \qquad \frac{\sigma, wo, k \longrightarrow \sigma', b_{\$()}, \text{error } e}{\sigma, wo, e, k\ w \longrightarrow \sigma, b_{\$()}, \text{error } e}\ \textsc{EWCtrlErr}$$

Control-code expansion $\boxed{\sigma, wo, k \longrightarrow \sigma, b_{\$()}, wer}$

$$\frac{(\sigma_1, \mathtt{fd}_r, \mathtt{fd}_w) = \mathtt{pipe}(\sigma_0) \quad (\sigma_2, \mathtt{pid}) = \mathtt{forkShell}(\sigma, c\ \mathtt{fd}_w\mathtt{>\&1\ fd}_r\mathtt{>\&\bullet}) \quad \sigma_3 = \mathtt{close}(\sigma_2, \mathtt{fd}_w)}{\sigma_0, wo, \mathtt{\$(}c\mathtt{)} \longrightarrow \sigma_3, \top_{\$()}, \langle \cdot, \text{cmdsubst } c\ \mathtt{pid\ fd}_r \rangle}\ \textsc{CmdSubst}$$

$$\frac{(\sigma_1, s) = \mathtt{readAll}(\sigma_0, \mathtt{fd}_r) \quad \sigma_2 = \mathtt{close}(\sigma_1, \mathtt{fd}_r)}{\sigma_0, wo, \text{cmdsubst } c\ \mathtt{pid\ fd}_r \longrightarrow \sigma_2, \top_{\$()}, \langle \cdot, \text{cmdwait } c\ \mathtt{pid\ trimRNL}(s) \rangle}\ \textsc{CmdSubstRead}$$

$$\frac{(\sigma_1, \text{error } e) = \mathtt{readAll}(\sigma_0, \mathtt{fd}_r)}{\sigma_0, wo, \text{cmdsubst } c\ \mathtt{pid\ fd}_r \longrightarrow \sigma_1, \top_{\$()}, \text{error } e}\ \textsc{CmdSubstReadErr}$$

$$\frac{(\sigma_1, n'_{\$?}) = \mathtt{wait}(\sigma_0, \mathtt{pid})}{\sigma_0, wo, \text{cmdwait } c\ \mathtt{pid}\ s \longrightarrow \sigma_1[n_{\$?} = n'_{\$?}], \top_{\$()}, \langle \text{exp } s, \cdot \rangle}\ \textsc{CmdSubstWait}$$

System call	**Description**
$\mathtt{pipe}(\sigma) = (\sigma', \mathtt{fd}_r, \mathtt{fd}_w)$	Create a FIFO pipe
$\mathtt{forkShell}(\sigma, c) = (\sigma, \mathtt{pid})$	Fork a new shell running c
$\mathtt{close}(\sigma, \mathtt{fd}) = \sigma$	Close a file descriptor
$\mathtt{readAll}(\sigma, \mathtt{fd}) = (\sigma', s \mid \text{error } e)$	Read all of a file descriptor until EOF
$\mathtt{waitpid}(\sigma, \mathtt{pid}) = (\sigma', n_{\$?})$	Wait for a PID

Helper function	**Description**
$\mathsf{trimRNL}(s) = s'$	Remove trailing newlines

Fig. 7. Small-step semantics for initial word expansion (expand) and command substitution

When the command has finished, we save its exit status and return the command's output as the expanded words exp s (CmdSubstWait). Note that each of the CmdSubst* rules sets $b_{\$()}$ to true.

Command semantics (negation) $\boxed{\sigma, b_\checkmark, c \longrightarrow \sigma, c}$

$$\frac{\sigma, \top_\checkmark, c \longrightarrow \sigma', c'}{\sigma, b_\checkmark, !\,c \longrightarrow \sigma', !c'}\ \textsc{Not} \qquad \frac{c \in \{\text{break } n, \text{continue } n, \text{return}, \text{exit}\}}{\sigma, b_\checkmark, !\,c \longrightarrow \sigma, c}\ \textsc{NotCtrl}$$

$$\frac{\sigma.n_{\$?} = 0}{\sigma, b_\checkmark, !\,\text{done} \longrightarrow \sigma[n_{\$?} = 1], \text{done}}\ \textsc{NotSuccess} \qquad \frac{\sigma.n_{\$?} \neq 0}{\sigma, b_\checkmark, !\,\text{done} \longrightarrow \sigma[n_{\$?} = 0], \text{done}}\ \textsc{NotFail}$$

Fig. 8. Small-step semantics for evaluation of negation

4.3 Evaluation

Shell programs are evaluated in a line-oriented fashion: a command is read and then immediately evaluated. The most interesting bits of evaluation are simple commands $(s{=}w)^*\ w_{\text{cmd}}\ r^*$, which make a variety of system calls. We give a detailed, formal semantics for them in Section 4.3.1, but two other command forms merit discussion: the `case` conditional and the `for` loop.

The `case` conditional has the form `case` w `in (`pat_1`)` s_1`;; ... esac`, where w are words to expand and then match against the patterns pat_i. The 'branch-on-pattern' style of conditional is a cousin of the `switch` conditional seen in imperative languages like C, though the shell uses string pattern matching instead of equality. Case analysis doesn't involve so many steps, but it involves a restricted form of expansion (stopping at arithmetic expansion) and pattern matching; the quoting rules for patterns have subtle interactions with the way dynamic pattern escaping (as opposed to parse-time escaping) is treated in the shell.

The `for` loop has the form `for` x `in` w`; do` c`; done`, where x is a variable name (without the `$`), w are words to be expanded, and c is a command. The words w are expanded to zero or more fields; c is executed once for each such field, with x bound to each field in turn. The shell's `for` loop is like a `for-each` loop in Java or a `for ... of` loop in JavaScript, where the iterated-over words are interpreted as a list according to field splitting and `$IFS`. Note that the shell's `for` loop expands the words before iterating, so there can be no confusion around concurrent modification (as in, e.g., Java).

We give a semantics to commands as a small-step relation $\sigma, b_\checkmark, c \longrightarrow \sigma', c'$, which we read as "in state σ, when $b_\checkmark$ indicates whether someone else is responsible for checking our exit status, the command c steps to a new state σ' and a new command c'". Our inference rules here are hand transcriptions of Smoosh's code. We estimate that Smoosh's evaluation semantics contains 100 additional evaluation rules not given here in the paper. Much of the evaluation relation is standard: control (`;`, `if`, `&&`, `||`, `!`, `while`, function definition) works more or less the usual way, though rather than having actual boolean values, the last exit status is looked up in the shell state.

As a warmup, we give the four rules for negation (Figure 8). The first rule is a standard congruence rule: the term $!\,c$ takes a step by stepping c itself (Not). Note that we set $\top_\checkmark$ in the premise. When the shell has the flag `errexit` set, we will exit on a non-zero exit status, but that behavior is proscribed "when executing the compound list following the `while`, `until`, `if`, or `elif` reserved word, a pipeline beginning with the `!` reserved word, or any command of an AND-OR list other than the last" ([Austin Group 2018], §4, `set`). Setting $\top_\checkmark$ tells subsidiary rules that someone else will be checking the exit status, and the shell should not exit on error even if `errexit` is set. The second rule propagates control: if the command c is one of the internal AST nodes used to represent control, then we must propagate the control command through the negation (NotCtrl). The final two rules do the actual work of negation, turning a successful exit status of 0 into 1 (NotSuccess)

and an unsuccessful nonzero exit status into 0 (NotFail). We show these four rules to assuage concerned readers: much of the semantics is standard.

Before offering a complete formal semantics for simple commands (Section 4.3.1), we mention the more interesting runtime forms necessary in Smoosh's semantics (Figure 4). The shell has a variety of line-oriented read/eval loops: the outermost shell itself, the 'dot' (`.`) a/k/a `source` command, and the `eval` command. In order to nest such loops, read/eval loops are part of Smoosh's AST: eval nodes represent an eval loop reading commands and eval_{cmd} nodes represent an eval loop executing a command. There are several other interesting AST forms: redirs holds on to information for restoring redirections; exec turns into a call to `execve`; wait turns into a call to `waitpid`; checked is used to disable `errexit` checking in subshells; and trapped and call are stack frames representing the current trap being processed and the current function being called, respectively.

4.3.1 Simple Commands. Here we highlight a key part of the shell's semantics: the evaluation of simple commands (Figures 9, 10, and 11). Assignments, commands, and redirections are agglomerated in the POSIX specification into a single form: a *simple command* of the form $(s{=}w)^*\ w_{\text{cmd}}\ r^*$, where the s are variable names, each w is a word for expansion either in an assignment or as the command and arguments itself (w_{cmd}), and each r is a redirection. Simple commands expand arguments, redirections, and then assignments—though assignments may be expanded before redirections if there are no arguments (i.e., $w_{\text{cmd}} = \cdot$). A five-step, twelve-point outline determines how to perform assignments (§2.9.1); a two-step, eleven-point outline determines how to find out if a command should be resolved as a special builtin, a builtin, a function, or an executable (§2.9.1 "Command Search and Execution"). Simple commands hardly live up to their name. In fact, they are the most complex part of command evaluation... and they may not even run a command!

In our model, evaluation of the simple command $(s{=}w)^*\ w_{\text{cmd}}\ r^*$ proceeds as follows. First, the words of the command arguments (w_{cmd}) are expanded from left to right (CmdStart, CmdArgExp, CmdArgErr, CmdArgDone). Next, redirections are expanded (CmdRedir, CmdRedirDoneErr) and applied (CmdRedirDone), saving file descriptor information *sfds* for later unwinding. Then we expand the words in the assignments (CmdAssign, CmdAssignErr) and store the results in a fresh local environment (CmdAssignSet). At this point, we are already in dangerous territory: the POSIX specification allows assignments to be expanded before redirections when there are no arguments. Our deterministic semantics always does redirections first; `bash` and `ksh` will run assignments first when there is no command.

Even in defining the relatively simple behavior of these expansions, there's been call for a variety of helper functions (Figure 10). Some of these helpers are quite simple—cmdSubst just updates the command options *co* to reflect whether expansion performed command substitution. Others have significant logic and make system calls: runCmd is 65 SLOC of Lem code that includes executable resolution using `$PATH` (via a separate, 50 SLOC function), a shell fork, and an exec form which will cause the semantics to call `execve`. The redirection functions redir and unredir make a variety of system calls for manipulating files (`fileRedir`, `closeAndSaveFD`, `renumberFD`, and `heredoc`).

Resuming our explanation of simple commands: after all of the expansion has been done (Figure 9), it is time to try to actually run the command (Figure 11). It may be that our simple command has no command at all, in which case we pop the local assignments off the stack and add them to the global environment $\sigma'.\rho$ (CmdAssignDoneNoCmd). The redirections are undone via unredir. Now $b_{\$()}$ pays off: if a command substitution was performed, we should return its exit status in $\sigma'.n_{\$?}$; if not, we should succeed with exit status zero. If, on the other hand, there is a command, then we step to a 'ready' state, capturing the assignments accumulated in the local environment. Our use of setLocal is a particular choice, not at all mandated by POSIX, which doesn't have local variables.

Command semantics (simple commands) $\boxed{\sigma, b_{\checkmark}, c \longrightarrow \sigma, c}$

$$\frac{}{\sigma, b_{\checkmark}, (s{=}w)^*\ w\ r^* \longrightarrow \sigma, (s{=}w)^*\ (\text{start}\ \langle \top_{\text{split}}, \top_{\text{glob}} \rangle\ w)\ r^*\ \langle \bot_{\text{cmdsubst}}, \top_{\text{fork}}, \bot_{\text{simple}} \rangle}\ \textsc{CmdStart}$$

$$\frac{\sigma, es \longrightarrow \sigma', b_{\$()}, es' \qquad es' \neq \text{error}\ f \qquad es' \neq \text{done}\ f}{\sigma, b_{\checkmark}, (s{=}w)^*\ es\ r^*\ co \longrightarrow \sigma', \text{cmd}_{\text{args}}\ (s{=}w)^*\ es'\ r^*\ \text{cmdSubst}(co, b_{\$()})}\ \textsc{CmdArgExp}$$

$$\frac{\sigma, es \longrightarrow \sigma', \text{error}\ f}{\sigma, b_{\checkmark}, \text{cmd}_{\text{args}}\ (s{=}w)^*\ es\ r^*\ co \longrightarrow \text{expError}(\sigma', f, \top_{\text{exit}})}\ \textsc{CmdArgErr}$$

$$\frac{\sigma, es \longrightarrow \sigma', \text{done}\ f}{\sigma, b_{\checkmark}, \text{cmd}_{\text{args}}\ (s{=}w)^*\ es\ r^*\ co \longrightarrow \sigma', \text{cmd}_{\text{redirs}}\ (s{=}w)^*\ f\ \langle \cdot, \bullet, r^* \rangle\ co}\ \textsc{CmdArgDone}$$

$$\frac{\sigma, rs \longrightarrow \sigma', b_{\$()}, rs'}{\sigma, b_{\checkmark}, \text{cmd}_{\text{redirs}}\ (s{=}w)^*\ f\ rs\ co \longrightarrow \sigma, \text{cmd}_{\text{redirs}}\ (s{=}w)^*\ f\ rs'\ \text{cmdSubst}(co, b_{\$()})}\ \textsc{CmdRedir}$$

$$\frac{\sigma, rs \longrightarrow \sigma', \text{error}\ f}{\sigma, b_{\checkmark}, \text{cmd}_{\text{redirs}}\ (s{=}w)^*\ f\ rs\ co \longrightarrow \text{expError}(\sigma', f, \text{special}(f)_{\text{exit}})}\ \textsc{CmdRedirErr}$$

$$\frac{\begin{array}{c}\text{redir}(\sigma, er^*) = (\sigma', f) \\ b_{\text{exit}} = (\neg b_{\checkmark} \wedge \texttt{errexit} \in \sigma'.\text{opts}) \\ \vee\ (\text{special}(f_{\text{cmd}}) \wedge \neg co.b_{\text{simple}} \wedge \neg\text{interactive}(\sigma')))\end{array}}{\sigma, b_{\checkmark}, \text{cmd}_{\text{redirs}}\ (s{=}w)^*\ f_{\text{cmd}}\ \langle er^*, \bullet, \cdot \rangle\ co \longrightarrow \text{redirError}(\sigma', f, b_{\text{exit}})}\ \textsc{CmdRedirDoneErr}$$

$$\frac{\text{redir}(\sigma, er^*) = (\sigma', sfds)}{\begin{array}{c}\sigma, b_{\checkmark}, \text{cmd}_{\text{redirs}}\ (s{=}w)^*\ f\ \langle er^*, \bullet, \cdot \rangle\ co \longrightarrow \\ \sigma'[\ell^* = \sigma'.\ell\ \cdot], \text{cmd}_{\text{assigns}}\ (s{=}\text{start}\ w\ \langle \bot_{\text{split}}, \bot_{\text{glob}} \rangle)^*\ f\ sfds\ co\end{array}}\ \textsc{CmdRedirDone}$$

$$\frac{\sigma, es \longrightarrow \sigma', b_{\$()}\ es' \qquad es' \neq \text{error}\ f \qquad es' \neq \text{done}\ f}{\begin{array}{c}\sigma, b_{\checkmark}, \text{cmd}_{\text{assigns}}\ s{=}es\ (s{=}es'')^*\ f_{\text{cmd}}\ sfds\ co \longrightarrow \\ \sigma, \text{cmd}_{\text{assigns}}\ s{=}es\ (s{=}es'')^*\ f_{\text{cmd}}\ sfds\ \text{cmdSubst}(co, b_{\$()})\end{array}}\ \textsc{CmdAssign}$$

$$\frac{\sigma, es \longrightarrow \sigma', b_{\$()}, \text{error}\ f}{\sigma, b_{\checkmark}, \text{cmd}_{\text{assigns}}\ s{=}es\ (s{=}es'')^*\ f_{\text{cmd}}\ sfds\ co \longrightarrow \text{expError}(\sigma, f, \top_{\text{exit}})}\ \textsc{CmdAssignErr}$$

$$\frac{\begin{array}{c}\sigma, es \longrightarrow \sigma', b_{\$()}, \text{done}\ f \qquad \text{setLocal}(\sigma', s, \text{toString}(f)) = \text{error}\ f_{\text{err}} \\ b_{\text{exit}} = (\neg b_{\checkmark} \wedge \texttt{errexit} \in \sigma'.\text{opts}) \vee \neg\text{interactive}(\sigma')\end{array}}{\sigma, b_{\checkmark}, \text{cmd}_{\text{assigns}}\ s{=}es\ (s{=}es'')^*\ f_{\text{cmd}}\ sfds\ co \longrightarrow \text{expError}(\sigma, f_{\text{err}}, b_{\text{exit}})}\ \textsc{CmdAssignSetErr}$$

$$\frac{\sigma, es \longrightarrow \sigma', b_{\$()}, \text{done}\ f \qquad \text{setLocal}(\sigma', s, \text{toString}(f)) = \sigma''}{\begin{array}{c}\sigma, b_{\checkmark}, \text{cmd}_{\text{assigns}}\ s{=}es\ (s{=}es'')^*\ f_{\text{cmd}}\ sfds\ co \longrightarrow \\ \sigma'', \text{cmd}_{\text{assigns}}\ (s{=}es'')^*\ f_{\text{cmd}}\ sfds\ co\end{array}}\ \textsc{CmdAssignSet}$$

Fig. 9. Small-step semantics for simple commands (part 1: expansion)

Helper function	**Description**
$\mathsf{special}(f) = b$	Determine if f is a special builtin
$\mathsf{setParam}(\sigma, x, s) = \sigma' \mid \mathsf{error}\ s$	Set a parameter
$\mathsf{setLocal}(\sigma, x, s) = \sigma' \mid \mathsf{error}\ s$	Set a local parameter in the outermost scope
$\mathsf{cmdSubst}(co, b_{\$()}) = co'$	Disjunctively update $co.b_{\$()}$
$\mathsf{expError}(\sigma, f, b_{\mathsf{exit}}) = (\sigma', c)$	Possibly exit with an expansion error
$\mathsf{redirError}(\sigma, f, b_{\mathsf{exit}}) = (\sigma', c)$	Possibly exit with a redirection error
$\mathsf{redir}(\sigma, er^*) = (\sigma', sfds \mid f_{\mathsf{err}})$	Perform a redirection, recording saved FDs
$\mathsf{unredir}(\sigma, sfds) = \sigma'$	Restore saved FDs
$\mathsf{runCmd}(\sigma, b_{\swarrow}, \rho_{\mathsf{cmd}}, s, f, co)$ $= (\sigma', c, b_{\mathsf{restore}}) \mid (\sigma', \mathsf{error}\ f_{\mathsf{err}})$	Resolve command (builtin, function, executable)
$\mathsf{toAssigns}(\ell) = \rho$	Convert a local environment to bindings
$\mathsf{mayExit}(b_{\mathsf{exit}}, c) = c'$	Conditionally exit
$\mathsf{checkTraps}(\sigma, c) = (\sigma', c')$	Check for pending signals and load trap handlers

System calls for runCmd	**Description**
$\mathtt{fileExists}(\sigma, s) = b$	Determine if a file exists
$\mathtt{fileExecutable}(\sigma, s) = b$	Determine if a file is executable
$\mathtt{execve}(\sigma, s_{\mathsf{cmd}}, s_{\mathsf{argv}[0]}, s^*_{\mathsf{argv}}, \rho, b_{\mathsf{sh}})$ $= (\sigma', c \mid \mathsf{error}\ s)$	Replace the current process

System calls for redir **and** unredir	**Description**
$\mathtt{fileRedir}(\sigma, ft, s) = (\sigma', \mathtt{fd} \mid \mathsf{error}\ s)$	Open a file redirection
$\mathtt{closeAndSaveFD}(\sigma, \mathtt{fd}) = (\sigma', sfds \mid \mathsf{error}\ s)$	Close an fd, saving it at an $\mathtt{fd} \geq 10$
$\mathtt{renumberFD}(\sigma, b_{\mathsf{close}}, \mathtt{fd}_{\mathsf{orig}}, \mathtt{fd}_{\mathsf{wanted}})$ $= (\sigma', sfds \mid \mathsf{error}\ s)$	Renumber an fd, saving the target
$\mathtt{heredoc}(\sigma, s) = (\sigma', \mathtt{fd} \mid \mathsf{error}\ s)$	Create an fd holding heredoc contents

System calls for checkTraps	**Description**
$\mathtt{pendingSignal}(\sigma) = (\sigma', \mathtt{sig}^?)$	Get a signal if one is pending

Fig. 10. Helpers for simple command execution

We use local environments because (a) we need quasi-local things to track assignments, and (b) it is convenient to reuse the architecture we built for local variables (see Section 8.3).

When faced with $\mathsf{cmd}_{\mathsf{ready}}$, there are three possibilities: we could have disabled execution (CmdRunNoexec); we could be running a special builtin, in which case POSIX mandates that any assignments be global (CmdRunSpecial); or we could be running an ordinary command (CmdRun). In the latter two cases, we step to the run form to actually run a command.

Finally, the run form triggers a call to the runCmd helper function. One of three things happens: we encounter an error of some kind, in which case noninteractive shells may exit (RunFail); the command runs and produces a non-zero exit status, in which case the command may exit due to the `errexit` option (RunErr); or, the command may run with no need to exit, in which case we continue with the command returned from runCmd (Run). The new command c yielded by runCmd will be one of three things: done, because a builtin was run to completion; a call form, because a function was called; or a wait form, because an executable was forked but should run in

Command semantics (simple commands, continued) $\boxed{\sigma, b_\checkmark, c \longrightarrow \sigma, c}$

$$\frac{\sigma.\ell^* = \ell_0^*\ \ell_{\mathsf{cmd}} \qquad n'_{\$?} = \begin{cases} \sigma.n_{\$?} & co.b_{\$()} \\ 0 & \text{otherwise} \end{cases} \qquad \sigma' = \mathsf{unredir}(\sigma, \mathit{sfds})[\ell^* = \ell_0^*][\rho = \sigma.\rho[\ell_{\mathsf{cmd}}][n_{\$?} = n'_{\$?}]]}{\sigma, b_\checkmark, \mathsf{cmd}_{\mathsf{assigns}}\ \cdot\ \cdot\ \mathit{sfds}\ co \longrightarrow \mathsf{xtrace}(\sigma', \ell_0^*), \mathsf{done}}\ \textsc{CmdAssignDoneNoCmd}$$

$$\frac{\sigma.\ell^* = \ell_0^*\ \ell_{\mathsf{cmd}}}{\sigma, b_\checkmark, \mathsf{cmd}_{\mathsf{assigns}}\ \cdot\ s\ f\ \mathit{sfds}\ co \longrightarrow \mathsf{xtrace}(\sigma, \ell_{\mathsf{cmd}}\ s\ f), \mathsf{cmd}_{\mathsf{ready}}\ \mathsf{toAssigns}(\ell_{\mathsf{cmd}})\ s\ f\ \mathit{sfds}\ co}\ \textsc{CmdAssignDoneCmd}$$

$$\frac{\mathsf{noexec} \in \sigma.\mathsf{opts}}{\sigma, b_\checkmark, \mathsf{cmd}_{\mathsf{ready}}\ \rho\ s_{\mathsf{cmd}}\ f\ \mathit{sfds}\ co \longrightarrow \sigma, \mathsf{done}}\ \textsc{CmdRunNoexec}$$

$$\frac{\mathsf{noexec} \notin \sigma.\mathsf{opts} \qquad \mathsf{special}(s) \qquad \neg co.b_{\mathsf{simple}} \qquad \sigma' = \sigma[\rho = \sigma.\rho \cup \rho_{\mathsf{cmd}}]}{\sigma, b_\checkmark, \mathsf{cmd}_{\mathsf{ready}}\ \rho_{\mathsf{cmd}}\ s\ f\ \mathit{sfds}\ co \longrightarrow \sigma', \mathsf{run}\ \rho_{\mathsf{cmd}}\ s\ f\ \mathit{sfds}\ co}\ \textsc{CmdRunSpecial}$$

$$\frac{\mathsf{noexec} \notin \sigma.\mathsf{opts} \qquad \neg\mathsf{special}(s) \vee co.b_{\mathsf{simple}}}{\sigma, b_\checkmark, \mathsf{cmd}_{\mathsf{ready}}\ \rho_{\mathsf{cmd}}\ s\ f\ \mathit{sfds}\ co \longrightarrow \sigma, \mathsf{run}\ \rho_{\mathsf{cmd}}\ s\ f\ \mathit{sfds}\ co}\ \textsc{CmdRun}$$

$$\frac{\mathsf{runCmd}(\sigma, b_\checkmark, \rho_{\mathsf{cmd}}, s, f, co) = (\sigma', \mathsf{error}\ f_{\mathsf{err}}) \qquad b_{\mathsf{exit}} = (\neg b_\checkmark \wedge \mathtt{errexit} \in \sigma'.\mathsf{opts}) \vee (\mathsf{special}(s) \wedge \neg co.b_{\mathsf{simple}} \wedge \neg\mathsf{interactive}(\sigma'))}{\sigma, b_\checkmark, \mathsf{run}\ \rho_{\mathsf{cmd}}\ s\ f\ \mathit{sfds}\ co \longrightarrow \sigma', \mathsf{checkTraps}(\mathsf{mayExit}(b_{\mathsf{exit}}, \mathsf{redirs}\ \mathsf{done}\ \mathit{sfds}))}\ \textsc{RunFail}$$

$$\frac{\mathsf{runCmd}(\sigma, b_\checkmark, \rho_{\mathsf{cmd}}, s, f, co) = (\sigma', c, b_{\mathsf{restore}}) \qquad \sigma'.n_{\$?} \neq 0 \qquad \neg b_\checkmark \qquad \mathtt{errexit} \in \sigma'.\mathsf{opts}}{\sigma, b_\checkmark, \mathsf{run}\ \rho_{\mathsf{cmd}}\ s\ f\ \mathit{sfds}\ co \longrightarrow \sigma', \mathsf{exit}}\ \textsc{RunErr}$$

$$\frac{\mathsf{runCmd}(\sigma, b_\checkmark, \rho_{\mathsf{cmd}}, s, f, co) = (\sigma', c, \top_{\mathsf{restore}}) \qquad \sigma'.n_{\$?} = 0 \vee \neg(\neg b_\checkmark \wedge \mathtt{errexit} \in \sigma'.\mathsf{opts}) \qquad c' = \begin{cases} \mathsf{redirs}\ c\ \mathit{sfds} & b_{\mathsf{restore}} \\ c & \text{otherwise} \end{cases}}{\sigma, b_\checkmark, \mathsf{run}\ \rho_{\mathsf{cmd}}\ s\ f\ \mathit{sfds}\ co \longrightarrow \sigma', c'}\ \textsc{Run}$$

Fig. 11. Small-step semantics for simple commands (part 2: evaluation)

the foreground, and so it needs to be waited on. These forms will in almost all cases be wrapped with a redirs form; the `exec` special builtin is an exception, since its redirections have global effect. The redirs form calls unredir to restore the saved file descriptors *sfds*, as in CmdAssignDoneNoCmd.

It is the shell's responsibility to (a) receive and record signals as they arrive so that (b) those pending signals can be periodically checked and handled. We check for traps after completing the evaluation of commands. The checkTraps helper function uses the `pendingSignal` system call to track these signals and interrupt the main line of execution with trap handlers when necessary. Of the three Run* rules, only one rule actually checks traps: RunFail. RunErr need not check traps

because the shell is exiting. Run need not check traps because all three of the possible forms from runCmd will check traps when they complete.

5 SMOOSH'S IMPLEMENTATION

Having sketched in mathematical notation the Smoosh semantics, we turn our attention to Smoosh's actual executable semantics, i.e., the code. Smoosh is open source under the MIT license.[3] Smoosh is written primarily in Lem [Mulligan et al. 2014], an ML-like language that extracts to OCaml and Coq, among others. Smoosh uses dash's parser via `libdash`,[4] a library that tracks dash's main repository[5] but has hooks for accessing the parser and a few other functions, along with OCaml bindings to the dash AST. Smoosh is implemented across 20 files comprising 10 814 SLOC, of which 9 305 lines are in Lem and 1 509 are in OCaml. By way of comparison, dash has 14 633 SLOC, of which 13 318 are C code, 122 are shell scripts, and 1 193 are header files. The Smoosh code contains considerable extra material (about 1k SLOC): shim code for reading the dash AST and for rendering Smoosh ASTs in JSON for the stepper (Section 6). In terms of effort, Smoosh took approximately 1.5 person years to develop.

Smoosh's architecture consists of a core semantics with two configurable 'ports': one for the *driver*, which determines how the Smoosh semantics is used; and one for the *OS implementation*, which determines what system calls do and how the filesystem behaves. The core semantics is a small-step operational semantics, as sketched in Section 4. The semantics itself is not outrageously large (4 880 SLOC for the semantics and builtins, 2 605 SLOC for the OS interface, and 3 252 for AST definitions), with a roughly 3:1 ratio of supporting code to code in the two core stepping functions:

```
val step_expansion : forall α. OS α =>
 os_state α * expansion_state → expansion_step * os_state α * expansion_state
val step_eval : forall α. OS α =>
 os_state α → checking_mode → stmt → evaluation_step * os_state α * stmt
```

The `step_expansion` function (328 SLOC, the core expansion semantics) corresponds to the relation $\sigma, es \longrightarrow \sigma, b_{\$()}, es$ (Section 4.2); the `step_eval` function (706 SLOC, the core evaluation semantics) corresponds to the relation $\sigma, b_{\checkmark}, c \longrightarrow \sigma, c$ (Section 4.3). In addition to semantics for the POSIX shell language itself, we also provide implementations for all of the special builtins,[6] mandatory builtins,[7] and several others.[8]

The OS typeclass. The most interesting aspect of the Smoosh implementation is its `OS` typeclass. We have already introduced several system calls in the formal model (Figures 7 and 10). The system calls described there are part of 40 different calls that can be made to the operating system. We can break the OS interface into three areas of interest: true system calls (14 functions), used to work with processes, signals, and job control; file system calls (24 functions), used to traverse and manipulate the file system, where 10 of these calls correspond to POSIX `stat` and `lstat`; and parser interactions (2 functions), used to communicate values of `PS1` and `PS2` to the `libdash` parser. Note that the calls described here don't necessarily correspond to POSIX-defined functions or any particular operating system's interface. Some system calls correspond clearly (e.g., `execve`); other

[3] https://github.com/mgree/smoosh
[4] https://github.com/mgree/libdash
[5] https://git.kernel.org/pub/scm/utils/dash/dash.git/
[6] `break`, `:`, `continue`, `.` a/k/a `source`, `eval`, `exec`, `exit`, `export`, `readonly`, `return`, `set`, `shift`, `times`, `trap`, and `unset` [Austin Group 2018] §2.14
[7] `alias`, `bg`, `cd`, `command`, `false`, `fc`, `fg`, `getopts`, `hash`, `jobs`, `kill`, `newgrp`, `pwd`, `read`, `true`, `umask`, `unalias`, and `wait` [Austin Group 2018] §2.9.1 part 1(d)
[8] `echo`, `help`, `history`, `local`, `printf`, `test` a/k/a `[`, `type`.

Smoosh system calls (e.g., `heredoc`) correspond to several system calls (create a pipe, possibly spawn a process if the heredoc string is bigger than the OS buffer size, write the heredoc text to the pipe). Some other system calls don't correspond to true "system calls" in any sense at all: `pendingSignal` doesn't actually make any system calls, but merely looks at a data structure. We put `pendingSignal` in the OS typeclass because the nature of that signal-tracking data structure depends on the OS typeclass instance.

We have implemented two instances of the OS typeclass: the `system` implementation makes real system calls to the host OS, allowing us to use Smoosh as a real shell; the `symbolic` implementation makes no real system calls to the host OS but instead simulates a POSIX OS and filesystem. We use the `symbolic` instance in our stepper (Section 6). We can imagine other OS instances, e.g., a `readonly` instance that allows filesystem reads but neither writes nor `execve`. One can think of an OS instance as a capability to (possibly virtualized) OS resources [Dennis and Van Horn 1966].

The OS typeclass forces us to confront difficult issues in program structure and API design. We can see the root of the issue in `waitpid`. Different implementations of `waitpid` must do drastically different things. In `system` mode, we expect `waitpid` to actually call `wait4` and to block until the given PID has terminated. In `symbolic` mode, we expect the current symbolic process to suspend while another symbolic process proceeds. That is, in `system` mode the OS typeclass is just a shim between Smoosh and the host OS, and we can rely on the host OS's scheduler entirely. In `symbolic` mode, the OS typeclass must actually implement the scheduler itself (see Section 6.1, "Scheduling").

5.1 Limitations and challenges

We have not attempted to implement any of the POSIX locale functionality. It is a longer term goal to give a precise formal account of locales. For now, however, Smoosh uses OCaml functions for ordering strings and formatting numbers, which are locale-independent as of 2018. Smoosh's handling of the various terminal/TTY functions is incomplete.

Parsing. Smoosh currently supports only the `libdash` parser, as extracted from `dash`. The dash parser is certainly "good enough", as `dash` is the default `/bin/sh` on Debian and Ubuntu systems. Using its parser in Smoosh has some drawbacks, though. First, prompting using `PS1` and `PS2` is built in to `dash`'s parser. These prompt variables ought to be subject to variable expansion each time they are displayed—and `dash` will (incorrectly) use its own, internal expansion routine and environment to expand these variables. Second, the `dash` parser doesn't support common extensions to the POSIX specification, like statically parsed tests with the `[[` form. Third, `dash`'s lexer doesn't correctly support multi-byte characters, making it difficult to work with character sets like Unicode. Fourth and finally, `dash`'s parser is written in C and may therefore result in memory errors or security vulnerabilities. We are interested in connecting Smoosh to other parsers, like Morbig [Régis-Gianas et al. 2018] and the bash-compatible parser from `OSH` [Chu 2019].

Design challenges. Some of the biggest difficulties encountered developing Smoosh were design related, e.g., finding the right interface so the OS typeclass can have both system and symbolic implementations. Other difficulties came from trying to coordinate system calls in a small-step semantics (we came up with exec and wait AST nodes after several "close, but no cigar" attempts). Signal handling was another area that was particularly difficult to implement in all its nuances. We also faced more prosaic challenges with Lem (see below).

Some of our choices made implementation easier, though. For example, using small-step semantics meant we could trace very precisely what happened when something went wrong. Using immutable state made debugging easier. Having a symbolic mode and a stepper was also very useful.

Challenges with Lem. We chose to implement Smoosh in Lem so we could write one implementation that extracts to OCaml for execution and to Coq for proof. In retrospect, it would have perhaps

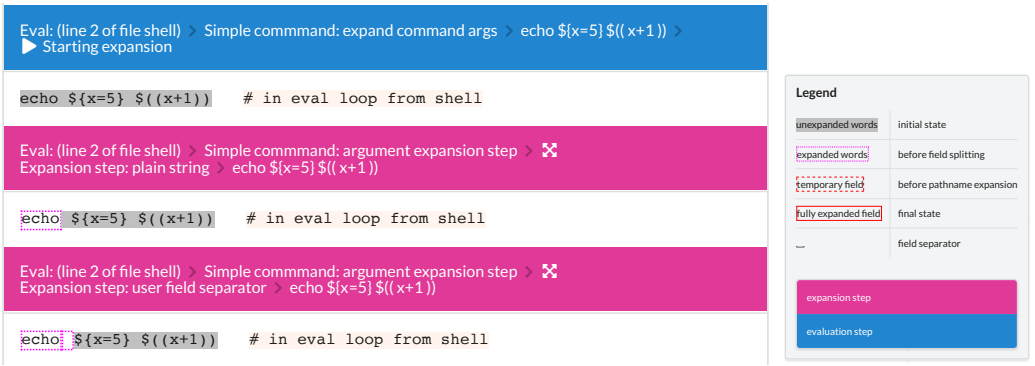

Fig. 12. A fragment of a trace from the Shtepper

been wiser to simply directly implement Smoosh in Coq. First, Lem's library hasn't yet mapped various concepts to Coq (e.g., the `uppercase_char` predicate). Second, Coq can extract to OCaml, too, and while Lem is well implemented, Coq is more robust and better supported. For example, we had to extend Lem to support character literals in pattern matching. When two branches of a pattern match disagree in return type, Lem simply gives the line numbers for the whole match—not particularly helpful for, e.g., `step_eval`, a 706-line match.

It's not possible to use a typeclass while instantiating it in Lem, so the actual `waitpid` system call takes `step_eval` as an argument. The `system` mode has no need for `step_eval`, but the `symbolic` mode relies on it (see Section 6.1). Our solution to this problem led to some redundancy: each instantiation of a typeclass defines the critical functions outside the instance, with whatever dependency or mutual recursion is necessary between operations. The typeclass instance definition then simply references these external definitions. Using Coq would not have helped with this issue: Coq does not allow you to use an instance while defining it, either.

6 SMOOSH'S SHELL STEPPER

We have used the Smoosh semantics to implement a *program stepper*, which we call the Shtepper. The Shtepper traces shell programs in a simulated environment. We use lightweight symbolic execution: symbolic values are treated symbolically as much as possible, but we don't support branches, i.e., we only explore a single path. When branching on a symbolic value, the current implementation will either treat the value as empty or will default to a sentinel value.

The Shtepper is implemented in two parts: a tracer and a visualizer. The tracer takes a shell program and a description of the environment in which to run it and produces a trace in JSON. The visualizer takes a JSON trace and displays it in a browser using JavaScript. The Shtepper is publicly available online.[9] The tracer comprises 1.2k SLOC of Lem and OCaml, implementing a synthetic POSIX environment (processes and filesystem), JSON mappings for the Smoosh AST, and a driver for the semantics. The visualizer comprises 1.5k SLOC of Ruby, ERB templates, JavaScript, and CSS; the bulk of the code (1.2k SLOC) is in the JavaScript trace rendering logic.

The visualizer. The Shtepper shows the main shell's thread of execution, highlighting evaluation steps in blue and expansion steps in pink (Figure 12). The shell contains a great deal of state and performs its work in many small steps. It is difficult to know what to highlight, what to merely show, and what to hide. By default, we show each step of expansion and evaluation, showing the

[9] http://shell.cs.pomona.edu/shtepper

environment, STDOUT, and STDERR at the beginning, after changes, and at the end of execution. We plan to study which information best helps novices learn the shell and avoid pitfalls.

6.1 Simulating POSIX

Our stepper runs in a simulated POSIX environment, using the `symbolic` OS instance. We use 'fuel' to limit the extent of symbolic execution. The web interface puts a conservative limit on fuel; at the risk of nontermination, the fuel can be made infinite in the tracer.

The symbolic shell state tracks the entirety of the POSIX environment: a list of FIFO pipes; a list of processes; the root of the symbolic filesystem; the current shell's mapping of file descriptors to pipes and files; whether or not the current shell has exited; the current shell's umask; and the contents of `/etc/passwd` to simulate calls to `getpwnam` during tilde expansion. Our symbolic model is sufficient to run interesting pipes between subshells, but insufficient to run executables.

Scheduling. When simulating a symbolic POSIX system, there is a tension between faithfully modeling all possible interleavings of processes and offering concise, legible information to the user of the stepper. To motivate the question, consider the following two pipelines:

(1) `while true; do echo 5; done | true`
(2) `while true; do echo 5; done | { read x; echo $((x+42)); }`

Both pipelines spawn two processes, both of which use shell builtins exclusively: neither of these pipelines needs to make an `execve` system call (though some systems may implement `true` or `echo` as executables, Smoosh and most shells build them in). In both cases, the first process will send `5` on STDOUT infinitely many times. In pipeline (1), the second process ignores its input entirely, terminating immediately. In pipeline (2), the second process reads a line of input, emits the line `47`, and terminates.

In a real POSIX system, both processes will be scheduled concurrently in both pipelines. The `while` loop will write as fast as it can until the pipe between the two processes is full, at which point the looping process will block. Pipeline (1)'s second process will terminate nearly immediately; pipeline (2)'s second process will terminate right after it's been able to read one line of input from the pipe. In either case, when the second process terminates, the pipe between the two processes has no more readers, and a `SIGPIPE` will be sent to the first process, terminating it.

How should we simulate these pipelines? We could use real threads to run the simulation, resulting in a different schedule each time. Such an approach would be error prone and nondeterministic; it is more appealing to use deterministic, simulated threads. How, then, do we schedule our simulated threads? If we always run the pipeline left to right, then both of the pipelines above will block when the pipe buffers fill up (or diverge if the buffers are unbounded). If we always run the pipeline right to left, then pipeline (1) will terminate immediately but pipeline (2) will block, waiting to read from a process that's never scheduled. Naïve scheduling regimes are insufficient.

We prioritize determinism and clarity over faithfully exploring all possible schedules. Scheduling takes place in rounds; every process in the system will take a step every round (which may mean blocking). Our scheduler will try to step each process in creation order: the root shell will get to go first, and then the first subshell, and so on. But: whenever a shell waits on another process (e.g., because of a foreground command or a command substitution) or would block reading from a pipe, we take the opportunity to step the waited on or writing process, making the scheduling order *demand-driven*. Scheduling according to demand keeps our scheduling deterministic without rigidly following creation order.

When waiting for a command to terminate, we put no bound on the number of steps we're willing to wait—blocking is the right behavior.[10] When a process calls `waitpid` in `symbolic` mode,

[10]Note that this waiting still counts as fuel usage, and our symbolic execution may give up while waiting.

the waited-on process is looked up and stepped. For a blocking wait—like `waitpid` or `readAll`—our scheduling works well. When waiting for a writer to supply input, we allow batches of 10 steps at a time, since one need not read all of the input: the `read` builtin only needs to read a line, and we'll get a shorter trace if we step the processes only enough to produce as much output as we need.

Concretely, the `read` builtin uses the `readline` system call:

$$\texttt{readLine}(\sigma, \texttt{fd}, b_\backslash) = (\sigma', s \mid \text{blocked } \texttt{pid} \mid \text{error } s)$$

A call to `readLine` tries to read from the file descriptor `fd`, with $b_\backslash$ specifying whether a backslash can allow a line continuation. The Smoosh `readLine` makes a series of calls to the POSIX `read` primitive in `system` mode. Getting back a new state σ', there are three possible results: getting a string, blocking, or an error. How do these cases arise? It could be the case that reading from `fd` causes an error, e.g., it's not an active file descriptor, points to a directory, etc. It could be the case that `fd` has enough text in the (symbolic) buffer to read a whole line, either because an unescaped newline is actually present or because there are no more writers to `fd`, and so EOF has been reached. In either of these cases, no scheduling need happen. Finally, it could be the case that there isn't enough text to read a line but there are still active writers. In this circumstance, a real call to `read` would block. In `symbolic` mode, we return the `pid` of the first process that could write to `fd`; that process is stepped. We must be careful, though: we don't want to block forever on the writing process, only enough to finish reading the line. In this circumstance, we allow the process to only step a few times before checking the buffer again (hence the default of 10 steps).

Our scheduler is not a completely realistic model of POSIX scheduling—we don't get to see every possible interleaving. Our scheduling does, however, model the way pipes and processes interact well enough for us to simulate interesting shell pipelines. Our scheduler is acceptable in light of its benefits: simple engineering; API compatibility between `system` and `symbolic` modes; a straightforward, linear visualization of shell stepping; and determinism without rigidity.

Filesystem. Our model of the filesystem is quite simplistic: we track a hierarchy of files and directories, but not file contents. Such a simple model suffices to simulate pathname expansion and file descriptor redirections (`dup`) and heredoc redirections (`here`), but not file redirections (`file`). It is a matter of engineering effort to produce a better symbolic filesystem. It would be interesting to link our symbolic system up with SibylFS (also written in Lem) [Ridge et al. 2015] or Forest [Fisher et al. 2011], or to use Ntzik et al.'s reasoning [2017; 2018]. We can also imagine implementing a read-only filesystem that allows access to the real, underlying filesystem, but treats writes (and other dangerous operations, like `execve`) as noops. We consider platform-specific filesystems, like `/proc`, as out of scope; we could in theory apply platform-dependent reasoning [Nita et al. 2008].

7 POSIX CONFORMANCE

Smoosh is meant to serve as a formal foundation for the POSIX shell. But is Smoosh a good model? We use three execution-based test suites to test Smoosh for conformance: the official POSIX test suite, the Modernish test suite, and a suite of our own devising. What are these test suites, why does using them support our claim, and how well do we conform?

The POSIX test suite. The Open Group VSC test suite[11] is a broad set of tests for POSIX conformance. We use VSC-PCTS2016 version 2.15. The POSIX shell test suite has 494 tests: 1 is 'not in use', 31 are 'untestable' and are checked by hand during certification, and 44 are locale-dependent and marked 'unresolved' or 'unsupported'. There are 418 locale-independent tests; these are the tests we use to compare against other shells. The POSIX test suite is not generally publicly available, and so most shells are not tested against it; we thank the Open Group for granting us a license.

[11] https://www.opengroup.org/testing/testsuites/vscpcts2003.htm

The Modernish test suite. Modernish is a substantial library for the POSIX shell [Dekker 2019] aimed at simplifying the shell language. It uses existing shell features and compatibility testing to construct a new, shell-like language on top of an existing shell. Modernish is implemented in 12k SLOC of shell scripts. It is very portable. To achieve this portability, Modernish runs a series of diagnostics against its host shell, detecting a variety of bugs and quirks.

Our test suite. Our third test suite is our own. Our test suite has two parts: internal unit tests and external system tests. Both parts serve as regression tests, but we can only use the external system tests on other shells, as the internal unit tests run Smoosh subsystems in symbolic mode. Each external system test pairs a short shell script with expected output and exit status. These are "whole system tests" exercising obscure corners of the shell's behavior. A test harness runs the scripts and checks outputs for a given shell executable. There are 161 external system tests, in five categories: 2 speed tests, 67 tests of builtin commands, 2 parsing tests, 82 tests of shell semantics, and 8 tests of the `sh` executable's interface. Some of the external system tests are adapted from bugs found by the POSIX test suite and Modernish. Our test suite is occasionally too picky, demanding particular STDERR text when any diagnostic message would suffice or a particular exit status when any non-zero exit status would be conformant.

Why use these tests? We use the POSIX test suite because it is the de facto standard of what POSIX compliance is. It is imperfect (see Section 7.1), but covers a great deal of ground. It is also a significant stress test of the shell, comprising 29k SLOC of tests in addition to 58k SLOC of shell code in the harness (which, by default, runs in the shell under test). The Modernish test suite condenses a great deal of knowledge and experience with real shells into a very small package. Modernish's shell scripts also rely on detailed characteristics of shell behavior—during development, the Modernish test suite exercised several bugs in Smoosh's semantics that the POSIX test suite did not detect. Finally, our test suite not only tests against our own regressions from development, but also highlights corner cases not covered in other test suites.

The test counts may seem low compared to other languages—the ES6 test suite contains more than 29 000 tests.[12] With so few tests, there is a risk of overfitting. However: first, we are undercounting: each part of the POSIX test suite addresses many behaviors, so the number of tests actually run is much higher. For example, a single test case in the POSIX suite concerning `set -u` actually tests 12 assertions. Second, there are at least two places where Smoosh deliberately overfits, in order to pass two outdated test cases carried over from an older version of the spec. The best solution is to have a broader test suite, including tests from, e.g., `OSH`, `ksh`, and NetBSD.

Conformance. We summarize the results of our various test suites in Table 1. Of all of the shells tested, Smoosh is the only one to have no failing tests in the POSIX test suite or in our own. In Modernish, Smoosh has no quirks and one bug: Smoosh's parser (borrowed from `dash`) cannot handle multibyte characters or characters with codes over 128; this bug triggers the one Modernish test that Smoosh fails.

7.1 Bugs found

We found several POSIX compliance bugs in our two primary reference shells, `dash` and `yash`, rediscovering a subtlety in the semantics for `printf` that had already been independently addressed. The results (Table 1) indicate that all shells other than Smoosh have other, nontrivial POSIX bugs, but we have not yet taken the time to diagnose and report each one. We also identified a typographical error in the POSIX specification and several bugs in the POSIX test suite. Identifying and reporting these bugs makes up only a small part of the dividends of formal semantics, but we mention them

[12] https://github.com/tc39/test262

	Smoosh 0.1	bash* 4.4-12(1)	dash 0.5.8-2.4	zsh† 5.3.1-4+b2	mksh 54-2+b4	ksh 93u+	yash* 2.43-1
POSIX test suite (418 tests)							
Failing tests	0	4 (8)	20	×	35	23	22 (23)
Time to run	12m41s	2m43s	2m43s	×	2m52s	3m24s	2m45
Modernish's shell diagnosis (91 potential bugs, 22 potential quirks) and test suite (312 tests)							
Bugs	1	16	2	3	3	14	1
Quirks	0	4	2	5	2	3	8
Failing tests	1	20	3	3	3	17	1
Time to run	5.5s	4.8s	1.4s	1.2s†	3.2s	2.2s	2.4s
Smoosh's test suite (161 tests)							
Failing tests	0	30 (35)	42	52	34	41	43 (39)
Time to run	23s	28s	1m13s	42s	21s	28s	29s

We use × to indicate that the tests could not be run. `OSH`'s results are omitted (see Section 7.1). *Both `bash` and `yash` initiate a strict POSIX mode when run as `/bin/sh`. The numbers in parentheses are the results from when strict POSIX mode is turned off. Timings are from POSIX mode. Modernish only uses POSIX mode. †`zsh` was run only in `emulate sh` mode, which Modernish uses as well. `zsh` crashes in the Modernish test suite when run noninteractively, so the timing is inaccurate.

Table 1. Comparison of shells on the POSIX test suite

to highlight that even before we've significantly applied the semantics, the process of development itself has been useful.

Bugs in shells. In `dash`, there were several issues: in arithmetic expansion, variables that were unset or empty were improperly treated; the `times` command reported incorrect numbers; and the empty alias was mishandled. We submitted patches for these bugs; the first was superseded by a different, independent fix of the same bug a year later; the second and third are under review. In `yash`, asynchronous commands (e.g., `curl ... &`) do not have their STDIN redirected to `/dev/null` and `fg` issues too much output. Other than the empty alias bug and the `fg` bug, which were caught by the POSIX test suite, all of the other bugs were detected by our own suite. Neither `zsh` nor `OSH` can run the POSIX test suite; `OSH` cannot run the Modernish suite, either, but it can run our suite; version 0.7.pre5 fails 39 tests and times out/hangs on 17 other tests.[13]

Bugs in the POSIX test suite and specification. We found ten bugs in the POSIX test suite, all of which have been confirmed as true bugs and will be fixed in the next version of the test suite. We also found typographical errors in the POSIX specification and in the POSIX test suite. Finally, we discovered a number of important shell behaviors that were *not* being tested. We are planning to submit our new tests to be added to the POSIX test suite.

7.2 Performance

Smoosh is substantially slower than existing implementations—about 4x slower on the POSIX test suite, slightly better in Modernish (Table 1).[14] The timings in our own test suite should not be taken too seriously: the only way to fail some of our tests is to hit a 9s time out; some shells even require manual intervention to finish running our tests of interactive features.

[13] https://www.oilshell.org/release/0.7.pre5/test/spec.wwz/smoosh.html

[14] Tests were run in Docker on an 2.8 GHz Intel Core i7 with 16GB RAM. Timings are from a single run, but there is little variation between runs.

Smoosh is slow because of our implementation strategy. Most shells are implemented as recursive evaluators in C, performing expansion by mutating compact data structures, whereas Smoosh is implemented as iterated small-step semantics in OCaml (via Lem), performing expansion by copying immutable, non-compact ASTs. Smoosh's slowness is not perceptible to us at the command line—interactive sessions spend the majority of their wall clock time running executables rather than the shell interpreter.

We speculated that some of Smoosh's slowness was because we weren't caching the filesystem calls in command name `$PATH` resolution, a process the specification calls 'hashing'. We implemented hashing, but our performance on the POSIX test suite was unchanged. Hashing may not be an optimization on modern systems.

8 BEYOND THE POSIX SPECIFICATION

The POSIX specification defines a common core for shells to implement, but every shell has to make decisions about what is left unspecified. Every shell we encountered implements extensions, too. In order for Smoosh's semantics to be a good model of the practical, implementation-oriented interpretation of the POSIX shell, we must understand how shells handle unspecified behavior, underspecified behavior, and extensions. We offer examples of each below.

When evaluating λ_{JS}, Guha et al. use the Mozilla test suite—they ensure that they get the same answers as Rhino, V8, and SpiderMonkey [Guha et al. 2010]. Implementations of the POSIX shell do not have nearly the same level of agreement as JavaScript interpreters do. When developing Smoosh, we compared to many other shells, which frequently disagree in corner cases. All of the shells implement unspecified extensions of the POSIX standard, too. Since none of these shells is perfectly POSIX compliant, we declined to precisely match any of them. We aimed instead for POSIX conformance, clear semantics, and a lack of quirks.

8.1 Unspecified behavior: non-local `break` and `continue`

Consider the case of non-lexical control. Should the following program print `hi` or not?

```
f() { break; echo hi; }; while true; do f; break; done
```

According to the POSIX specification, the `break` command in the function `f` is not lexically enclosed in the loop. Non-lexical use of the control builtins `break` and `continue` is left unspecified. Shells behave differently! The more sensible option is to forbid such non-lexical control, as most shells do, printing `hi` (and possibly a diagnostic message from the non-lexically enclosed `break` in `f`). But `bash` 3.2.57(1) (which comes with macOS) and `zsh` both allow the `break` to pass through the function call and exit the non-lexically enclosing loop. In Modernish, `zsh` and the old `bash`'s behavior is called a *quirk*. There is also ambiguity around the permissible behaviors of `break` and `continue` when used with a non-lexically enclosing loop or with no enclosing loop at all: the standard seems to forbid warning the user when control commands have no effect. The issue is under discussion with the Open Group.

Prioritizing safety and avoiding quirks means that Smoosh uses lexical control (and will print `hi`), though non-lexical control can be enabled with a flag (`set -o nonlexicalctrl`).

8.2 Underspecified behavior: `getopts` and hidden state

The POSIX specification mandates that the `getopts` builtin should use the user-visible shell variables `OPTIND` and `OPTARG` to parse command-line and function arguments. The rules are subtle, but the general idea is that `OPTIND` tracks the index of the current option argument; the `getopts` utility also takes the name of a variable to set with the current option. If `getopts` finds an option that takes an argument, the argument value is stored in `OPTARG`. There's a problem in the POSIX specification,

	Smoosh 0.1	bash 4.4-12(1)	dash 0.5.8-2.4	zsh 5.3.1-4+b2	OSH 0.6.pre21	mksh 54-2+b4	ksh 93u+	yash 2.43-1
nested scope (1)	✓	✓	✓	✓	✓	×	×	×
`local` (2)	special	builtin	special	reserved	builtin$^+$	special	×	×
readonly (3)	error	silent	error	override	override	override		
initial (4)	unset	unset	unset	null	unset	unset		
`-p` (5)	✓	✓	×	~	×	~		

(1) Do assignments before function calls have nested scope? (2) What kind of command is `local`? `zsh` identifies it as a reserved word; `OSH`'s `type` command calls it a builtin but it is in fact a syntactically restricted reserved word. Neither `ksh` nor `yash` support `local`. (3) If `x` is declared `readonly`, can a local `x` be defined? `bash` silently ignores the local definition; `zsh`, `OSH`, and `mksh` allow for a local override. (4) What is the initial value of `x` after running `local x` without assigning a value? (5) Both `readonly` and `export` will dump a list of variables when invoked with the argument `-p`; is there such a flag for `local`? `dash` does not implement `-p`. `bash` implements it, revealing that `local` is a macro for `declare`. Both `zsh` and `mksh` implement `local` as a macro for `typeset`, though `-p` also shows non-local variables.

Table 2. The `local` builtin and nested scope in different shells

though: `getopts` needs more information than `OPTIND` and `OPTARG` to work properly. In particular, `getopts` needs to keep track of the offset into the current argument to handle 'grouped' arguments.

As a concrete example, consider `getopts "ab:c:" opt -ab hi -c hello`, where `"ab:c:"` is the *optstring* specifying that `-a` is an option without arguments and that `-b` and `-c` are options that take arguments, `opt` is the variable name to set with the current option, and `-ab hi -c hello` is the argument list to process. After parsing the first option, `opt` ought to be `a`, but what should `OPTIND` be? And where should the shell record the fact that the `a` has already been processed? The choice we've seen taken in shells is to keep some extra state to record the offset of the next character to process inside of a grouped option. Every shell but `yash` keeps this state hidden; Smoosh stores it in the $\sigma.n^?_{\text{optoff}}$ variable. `yash` makes this state visible by adding the current offset as in `OPTIND=1:2`. Even so, shells differ slightly in their behavior, with some incrementing `OPTIND` earlier than others.

The issue has two root causes. First, the POSIX specification only *implies* that more state is needed. Second, the specification is silent on how to handle the implicit state, leading to divergent behaviors. The issue is under discussion on the Austin Group mailing list.

8.3 Unspecified behavior and extensions: scope and the `local` builtin

The POSIX shell has dynamic scope. Variable assignments on a command line, as in `LD_PRELOAD=... cmd bar baz` have three possible behaviors, depending on the nature of `cmd` (per Section 4.3). If `cmd` is a special builtin, then the assignments are globally visible. If `cmd` is a program or a non-special builtin, then the assignments are visible only to that program or builtin. Finally, if `cmd` is a function, then the assignments are visible during the dynamic extent of the function, but it is unspecified whether or not the assignments are globally visible afterwards: scope may or may not be 'nested'. Half of the shells we considered implement nested scope for functions (Table 2).

Relatedly, many shells implement a `local` builtin, which has syntax analogous to `export` and `readonly` (Table 2). That is, one can write `local x=5` inside of a function to declare a variable `x` that will be (globally) bound to `5` until the function returns, when `x` will revert to whatever value it had before `local` was used. Considering that the facilities to implement `local` line up nearly exactly with those needed for nested scope, it is unsurprising that of the four shells with nested scope, only

one (mksh) doesn't have local. OSH implements nested scope; our tests, however, revealed a bug in getopts along with a "serious bug" with scoping. OSH's bugs were fixed in a later release.

We implemented local in Smoosh, erring on the side of featureful safety. By analogy to export and readonly, we've treated local as a special builtin. Since erroneous conditions in special builtins cause shell scripts to abort, making this choice assures early failure in case local is misused. Like in dash, it is an error in Smoosh to create a local variable with the same name as a readonly variable. It may seem unduly restrictive, since the variable will be locally scoped, but triggering an error makes readonly variables unforgeable, even locally.

9 RELATED WORK

Research on the shell. The POSIX shell has seen relatively little academic attention. There are only two recent works that take the shell's semantics seriously: ABash [Mazurak and Zdancewic 2007] and CoLiS [Jeannerod 2017; Jeannerod et al. 2017a,b; Régis-Gianas et al. 2018]. ABash is a static analysis for bash scripts in particular; it checks for common expansion bugs along with taint tracking. ABash has some limitations in its parser and in its model of expansion; we hope that Smoosh's semantics can combine with ABash's approach to provide a more precise analysis that can cover more of the shell. The CoLiS project takes a core calculus approach to the shell, not unlike λ_{JS} [Guha et al. 2010]. Jeannerod et al. define an interpreter for a tiny shell-like language [2017; 2017a] to which they elaborate shell using their hand-built parser, Morbig [Régis-Gianas et al. 2018]. The CoLiS interpreter is not yet a usable shell—it passed only 8 of our 161 tests, mostly due to a variety of unsupported shell features (e.g., assignments in commands, heredocs, break); even so, its symbolic evaluator has already found numerous bugs in Debian maintainer scripts [Jeannerod et al. 2017b]. Smoosh has taken the "full semantics" approach rather than the "elaborate to a core calculus" approach; by implementing the full semantics, we have a baseline against which we can prove an elaboration correct. We believe it would have been *much* more challenging for us to achieve Smoosh's level of conformance if we had started with elaboration (see "Whole-language semantics" below). We have left parsing out of our scope for now (see Section 5.1); we hope to extend Smoosh to use Morbig and other parsers. We have already made arguments about why certain features of the shell are useful for concurrency [Greenberg 2018a] and interactivity [Greenberg 2018b]; see the latter paper for references to much older research on command-line interfaces, interactivity, and live programming [Collins et al. 2003]. NoFAQ [D'Antoni et al. 2016] uses machine learning to suggest repairs to commands, but treats shell syntax as unstructured text.

Tools for the shell. Outside of academia, programmers have created a variety of tools to support shell programming. Modernish is a library that rebuilds the shell language using its own features [Dekker 2019]. ShellCheck [koalaman 2016] is a linter that can process a variety of shell extensions; it is purely syntactic, though, and could be improved by having it reason using Smoosh's semantics. ExplainShell [Kamara 2016] patches together a parser for bash[15] and some post-processed man pages to explain each part of a shell command. ExplainShell isn't about semantic insight: e.g., asking it about ${#*} doesn't mention that its expansion is unspecified; nothing indicates the difference between $@ and "$@". Finally, maybe allows for tentative use of shell commands [Weidmann 2016].

Replacement shells, scripting languages, and shell libraries. A variety of replacement shells have been proposed, both in academia, with projects like Scsh, Shill, and Rash [Hatch and Flatt 2018; Moore et al. 2014; Shivers 2006] and outside of academia, with popular interactive shells like fish and zsh, along with less well known replacement shells like Xonsh and OSH [Chu 2019; Scopatz et al. 2019]. Only zsh and OSH aim for any real POSIX compatibility; Xonsh strives to have some measure of compatibility with bash. Others are more scripting languages than shells: Shivers planned but never

[15] https://github.com/idank/bashlex

developed an interactive mode for Scsh. Shill isn't meant to be interactive, either. Shell libraries, like Plumbum, Turtle, and Shcaml aim to bring shell-like idioms into conventional programming languages [Gonzalez 2018; Heller and Tov 2008; Schreiner et al. 2018]. None of these quite match our scope: POSIX shells that can be used both interactively and programmatically.

Whole-language semantics. There have been a variety of efforts at building language semantics for whole, real languages: for JavaScript [Guha et al. 2010; Maffeis et al. 2008], for C [Blazy and Leroy 2009; Ellison and Rosu 2012; Kang et al. 2015; Krebbers et al. 2014; Memarian et al. 2016], and for Rust [Jung et al. 2017; Weiss et al. 2018, 2019], to name a few. Approaches to such whole-language semantics fall broadly into two styles of modeling: large semantics that cover a broad range of language constructs, using light syntactic sugar; and small semantics that cover a narrow range of language constructs, using heavy syntactic sugar/elaboration. Lem and K epitomize the former style [Mulligan et al. 2014; Roşu and Şerbănuţă 2010], while λ_{JS} is emblematic of the latter style, defining a small calculus to which it elaborates JavaScript [Guha et al. 2010]. We chose to build a 'large' semantics for two reasons. First, the shell's complexity makes it hard to precisely encode every aspect of a shell construct in terms of other, simpler ones; moreover, using a large AST lets us better match parts of the semantics to the POSIX specification. Second, λ_{JS} leaves out `eval`—to include it requires including not only a parser in the desugarer or semantics, but also the desugarer itself. Later work on S5 does exactly that [Politz et al. 2012]. We cannot do without `eval` [Richards et al. 2011]: it's used in a variety of testing frameworks and is the only way to achieve certain forms of indirection in the shell (e.g., a tilde expansion with a variable as in Figure 1a). Now that we have much more experience with the shell—and a reference semantics against which to prove our elaboration correct—we can think about desugaring and compilation.

10 CONCLUSION

Smoosh is a small-step operational semantics for a new shell implementing the POSIX shell standard. The executable semantics is implemented in Lem code but corresponds well to legible, practicably complex inference rules. Of the shells we have considered, Smoosh best conforms to the POSIX specification and has the fewest bugs and quirks. In the process of developing and testing Smoosh, we found numerous bugs, confusions, and underspecifications in existing shells used in production, in the POSIX test suite, and in the POSIX specification itself.

Smoosh cleanly separates OS functionality from the core shell semantics, allowing us to use our tested semantics in symbolic settings; we demonstrate this capability with the Shtepper. The semantics we've developed here promises to support research on and development of shells and tools for shells. Our semantics will also form the basis of design innovations and revisions in the shell and, we hope, interactive programming in general.

ACKNOWLEDGMENTS

Much of this work was done while the first author was on sabbatical at Harvard University; he gratefully acknowledges the Harvard CS department in general and Steve Chong in particular for their support. Brian Selves of the Open Group provided invaluable assistance understanding the POSIX specification and the POSIX test suite, the latter of which the Open Group kindly provided a license for. The Austin Group mailing list offered helpful, timely feedback and discussion. Discussions with Ralf Treinen, Yann Régis-Gianas, and Andy Chu were informative and interesting. Shriram Krishnamurthi, Aaron Bembenek, and David Holland had useful comments. The POPL reviewers gave detailed and valuable feedback. Hannah de Keijzer proofread.

REFERENCES

The Austin Group. 2018. POSIX.1 2017: The Open Group Base Specifications Issue 7 (IEEE Std 1003.1-2008).

Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning* 43, 3 (01 Oct 2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

Andy Chu. 2019. Oil Shell. https://www.oilshell.org/

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

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

Martijn Dekker. 2019. Modernish. https://github.com/modernish/modernish

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9, 3 (March 1966), 143–155. https://doi.org/10.1145/365230.365252

Chucky Ellison and Grigore Rosu. 2012. An Executable Formal Semantics of C with Applications. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 533–544. https://doi.org/10.1145/2103656.2103719

Kathleen Fisher, Nate Foster, David Walker, and Kenny Q. Zhu. 2011. Forest: A Language and Toolkit for Programming with Filestores. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 292–306. https://doi.org/10.1145/2034773.2034814

Simson Garfinkel, Daniel Weise (Ed.), and Steven Strassman (Ed.). 1994. *The UNIX Hater's Handbook*. IDG Books Worldwide, Inc., San Mateo, CA, USA.

Gabriel Gonzalez. 2018. *Turtle: shell programming, Haskell style*. https://github.com/Gabriel439/Haskell-Turtle-Library

Michael Greenberg. 2018a. The POSIX shell is an interactive DSL for concurrency. DSLDI.

Michael Greenberg. 2018b. Word expansion supports POSIX shell interactivity. In *Programming Companion (presented at Programming eXperience (PX))*. ACM. https://doi.org/10.1145/3191697.3214336

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 126–150.

William Gallard Hatch and Matthew Flatt. 2018. Rash: from reckless interactions to reliable programs. In *Generative Programming: Concepts and Experiences (GPCE)*, Eric Van Wyk and Tiark Rompf (Eds.). ACM, 28–39. https://doi.org/10.1145/3278122.3278129

Alec Heller and Jesse A. Tov. 2008. Caml-Shcaml: an ocaml library for unix shell programming. In *ML*. ACM, 79–90.

Nicolas Jeannerod. 2017. Le coquillage dans le CoLiS-mateur. In *JFLA 2017 - Vingt-huitième Journées Francophones des Langages Applicatifs*. Gourette, France. https://hal.archives-ouvertes.fr/hal-01432034

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 (VSTTE) (Lecture Notes in Computer Science)*, Vol. 10712. Heidelberg, Germany. https://hal.archives-ouvertes.fr/hal-01534747

Nicolas Jeannerod, Yann Régis-Gianas, and Ralf Treinen. 2017b. Having Fun With 31.521 Shell Scripts. (April 2017). https://hal.archives-ouvertes.fr/hal-01513750 working paper or preprint.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

Idan Kamara. 2016. explainshell. http://explainshell.com/

Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A Formal C Memory Model Supporting Integer-pointer Casts. In *Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 326–335. https://doi.org/10.1145/2737924.2738005

koalaman. 2016. ShellCheck. https://github.com/koalaman/shellcheck/

Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. 2014. Formal C Semantics: CompCert and the C Standard. In *Interactive Theorem Proving*, Gerwin Klein and Ruben Gamboa (Eds.). Springer International Publishing, Cham, 543–548.

Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems (APLAS)*. Springer-Verlag, Berlin, Heidelberg, 307–325. https://doi.org/10.1007/978-3-540-89330-1_22

Karl Mazurak and Steve Zdancewic. 2007. ABASH: Finding Bugs in Bash Scripts. In *PLAS*. 105–114. https://doi.org/10.1145/1255329.1255347

Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the De Facto Standards. In *Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/2908080.2908081

Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. 2014. Shill: A Secure Shell Scripting Language. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX.

Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: Reusable Engineering of Real-world Semantics. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 175–188. https://doi.org/10.1145/2628136.2628143

Marius Nita, Dan Grossman, and Craig Chambers. 2008. A theory of platform-dependent low-level software. In *Principles of Programming Languages (POPL)*, George C. Necula and Philip Wadler (Eds.). ACM, 209–220. https://doi.org/10.1145/1328438.1328465

Gian Ntzik. 2017. *Reasoning About POSIX File Systems*. Ph.D. Dissertation. Imperial College London.

Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa Gardner. 2018. A Concurrent Specification of POSIX File Systems. https://doi.org/10.4230/LIPIcs.ECOOP.2018.4

Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. 2012. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Dynamic Languages Symposium (DLS)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/2384577.2384579

Yann Régis-Gianas, Nicolas Jeannerod, and Ralf Treinen. 2018. Morbig: A Static Parser for POSIX Shell. In *Software Language Engineering (SLE)*. Boston, United States. https://doi.org/10.1145/3276604.3276615

Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do. In *ECOOP*, Mira Mezini (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–78.

Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. 2015. SibylFS: Formal Specification and Oracle-based Testing for POSIX and Real-world File Systems. In *Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, USA, 38–53. https://doi.org/10.1145/2815400.2815411

Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

Henry Schreiner, Tomer Filiba, et al. 2018. *Plumbum: shell combinators*. http://plumbum.readthedocs.io/en/latest/

Anthony Scopatz et al. 2019. Xonsh. https://xon.sh/

Olin Shivers. 2006. SCSH manual 0.6.7. https://scsh.net/docu/html/man.html

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

Aaron Weiss, Daniel Patterson, and Amal Ahmed. 2018. Rust Distilled: An Expressive Tower of Languages. *CoRR* abs/1806.02693 (2018). arXiv:1806.02693 http://arxiv.org/abs/1806.02693

Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *CoRR* abs/1903.00982 (2019). arXiv:1903.00982 http://arxiv.org/abs/1903.00982