From Ahead-of- to Just-in-Time and Back Again: Static Analysis for Unix Shell Programs

Lukas Lazarek
lukas_lazarek@brown.edu
Brown University

Seong-Heon Jung
seong-heon_jung@brown.edu
Brown University

Evangelos Lamprou
evangelos_lamprou@brown.edu
Brown University

Zekai Li
zekai_li@brown.edu
Brown University

Anirudh Narsipur
anirudh_narsipur@brown.edu
Brown University

Eric Zhao
eric_c_zhao@brown.edu
Brown University

Michael Greenberg
michael@greenberg.science
Stevens Instisute of Technology

Konstantinos Kallas
kkallas@ucla.edu
UCLA

Konstantinos Mamouras
mamouras@rice.edu
Rice University

Nikos Vasilakis
nikos@vasilak.is
Brown University

ABSTRACT

Shell programming is as prevalent as ever. It is also quite complex, due to the structure of shell programs, their use of opaque software components, and their complex interactions with the broader environment. As a result, even when exercising an abundance of care, shell developers discover devastating bugs in their programs only at runtime: at best, shell programs going wrong crash the execution of a long-running task; at worst, they silently corrupt the broader environment in which they execute—affecting user data, modifying system files, and rendering entire systems unusable. Could the shell's users enjoy the benefits of semantics-driven static analysis before their programs' execution—as offered by most other production languages?

CCS CONCEPTS

• **Software and its engineering → Scripting languages**; *Compilers*; *Operating systems.*

KEYWORDS

Unix, Linux, shell, static analysis, type systems, inference

HOTOS 25, May 14–16, 2025, Banff, AB, Canada

ACM ISBN 979-8-4007-1475-7/25/05.
https://doi.org/10.1145/3713082.3730395

ACM Reference Format:
Lukas Lazarek, Seong-Heon Jung, Evangelos Lamprou, Zekai Li, Anirudh Narsipur, Eric Zhao, Michael Greenberg, Konstantinos Kallas, Konstantinos Mamouras, and Nikos Vasilakis. 2025. From Ahead-of- to Just-in-Time and Back Again: Static Analysis for Unix Shell Programs . In *Workshop in Hot Topics in Operating Systems (HOTOS 25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3713082.3730395

1 INTRODUCTION

Edition XVIII of HotOS featured a presentation on the Next 50 Years of the Shell [9], with a humorous segment in which a panel of experts opines on the right approach to improve the safety and correctness of shell scripts:

> *Static analysis! Type systems! More static analysis!*
> —exclaimed by experts [8]

The implied benefits are clear [17, 26]: fast pre-execution checks for computations that take days; detection of misbehaviors by a script's developer, not its user; improved error messages; whole-program optimization opportunities; and elimination of entire classes of bugs—well-typed shell programs cannot go wrong, at least not in their usual ways.

Yet this expert opinion comes to the presenters' dismay: the shell is uniquely challenging when it comes to anything static. Its pervasive dynamicity—*e.g.*, runtime evaluation, expansion, subshells—seems to make ahead-of-time checking intractable. Its tight integration with, and arbitrary effects on, the broader environment complicates reasoning about possible states and effects prior to program execution. And its opaque, polyglot commands—communicating with each

other and the shell through raw bytes—occlude deep reasoning. It is no accident that the shell, the oldest and most prevalent environment today [14], still relies on surface-level syntactic linting à la `ShellCheck` [5]. In the face of these challenges, the presenters argue for a shift from ahead-of-time to just-in-time analysis to enable shell rehabilitation.

This paper counter-argues: the shell can and should enjoy the benefits of ahead-of-time analyses, offered by other mainstream languages, by combining several insights.

First, *divide and conquer*. Disaggregate static guarantees into tractable subclasses, develop separate subsystems targeting each subclass, then reaggregate these subsystems. For example, constraints over command effects on the file system are expressible in Hoare-style logical pre- and post-conditions, whereas constraints over IPC and other stream contents are expressible in regular languages—offering benefits in computational tractability, succinctness, and familiarity for everyday Unix developers.

Second, *trust, but verify*. Leverage large-language models (LLMs) to translate documentation—the only common source of truth for opaque commands—into partial specifications, then interpose on commands to test these specifications. For example, the effects of a specific command invocation to the file system are inferable from its `man` page and checkable via file system containment and system-call tracing.

Third, *better late than sorry*. If ahead-of-time checking is insufficient to conclude safety, specification-aware runtime monitoring can stop execution before catastrophic bugs occur: a monitor can halt the execution of a script about to perform a dangerous action going against key invariants.

These insights can tame what the presenters call [9, §2.2] the shell's *bad* aspects—arbitrariness, dynamicity, obscurity—and eliminate its *ugliest*: error proneness (§2). They also open up additional opportunities for exciting future research on correctness and reliability—not only benefiting the shell, but also other prominent virtualization, containment, and cloud environments thinly wrapping shell constructs (§5). There is truly something here for everybody.

2 RENDERING SYSTEMS UNUSABLE

Consider the core of a bug in the Steam-for-Linux updater (Fig. 1), which famously wiped the file systems of several Steam users [33]. The updater first deletes the existing installation, whose location it identifies via a variable `STEAMROOT`. Its value is determined via runtime expansion in a subshell (`$(...)`). identify the path of the current script (`${0}`), expand it to remove anything after the last slash (`%/*`), change to that path (`cd...`), and report the current directory (`$PWD`). For some paths (*e.g.*, `~/.steam/upd.sh`), expansion results in the parent directory as intended (*e.g.*, `/home/jcarb/.steam`); for other paths (*e.g.*, ones lacking

```
1 #!/bin/sh
2 STEAMROOT="$(cd "${0%/*}" && echo $PWD)"
3 # ... more lines ...
4 rm -fr "$STEAMROOT"/*
```

Figure 1: The core of a Steam updater bug [33]. When expansion results in an empty `STEAMROOT` string (ln. 2), the script deletes everything user-writable (ln. 4).

any directories like `upd.sh`), expansion results in the script name, causing `cd` to fail and `STEAMROOT` to end up empty. The result: `rm -fr /*` deletes everything user-writable.

The states of the art and practice: Many earlier research efforts [4, 15, 20, 29] focus on the interaction of individual commands with the environment, not larger-scale program composition, or on a single composition primitive (*e.g.*, `|`). Tools such as ExplainShell [16], offer help with command invocations, but do not check correctness.

The most widely used tool is `ShellCheck` [5], a syntactic linter based on a collection of hard-coded patterns (*e.g.*, proper variable names).[1] The `ShellCheck` linter (v0.10.0) indeed issues a warning for Fig. 1, suggesting replacing `$STEAMROOT` with `"${STEAMROOT:?}"` to signal a runtime error if the variable is empty. Unfortunately, this kind of syntax-matching approach is limited: it fails to recognize an obviously safe fix (Fig. 2) and it fails to identify the unambiguous incorrectness of an obviously unsafe fix (Fig. 3). (The two attempted fixes, placed next to each other, differ by only one character.) Surface-level syntactic linting, while useful, is inherently noisy and context-insensitive.

3 SEMANTICS-DRIVEN ANALYSIS

Reasoning deeply about semantics at the scale and complexity of real shell programs requires several key ingredients.

Reasoning about state: A static analysis system for the shell must first be able to understand and reason about the state of the shell and its broader environment—*e.g.*, the file system, command arguments, and environment variables. The first ingredient is therefore to *generate and track relevant constraints on state*. To identify the bug in Fig. 1 (and recognize its absence in Fig. 2), the analysis needs to generate constraints for all variables and their contents. Examples of such variables include Fig. 1's `$0` and `$PWD`, whose contents may be file or directory paths. Such constraints can be captured by existing and well-understood formalisms—*e.g.*, by a regular expression of the form `/?([^/]*/)*[^/]+`. Such formalisms offer several benefits—*e.g.*, computational efficiency and ease-of-use for developers versed with the expressions found pervasively in the Unix environment.

[1] A quick exploration reveals that `ShellCheck` is used by about 65% of top GitHub shell repositories with over 1K shell LoC—which speaks volumes about the community's needs for static shell-script analysis.

```sh
#!/bin/sh
STEAMROOT="$(cd "${0%/*}" && echo $PWD)"

if [ "$(realpath "$STEAMROOT/")" != "/" ]; then
  rm -fr "$STEAMROOT"/*
else
  echo "Bad script path: $0"; exit 1
fi
```

Figure 2: An obviously safe fix to the Steam bug (*cf. Fig. 1*). The `rm -fr` line will *never* delete from the root—guaranteed across all executions and environments.

```sh
#!/bin/sh
STEAMROOT="$(cd "${0%/*}" && echo $PWD)"

if [ "$(realpath "$STEAMROOT/")" = "/" ]; then
  rm -fr "$STEAMROOT"/*
else
  echo "Bad script path: $0"; exit 1
fi
```

Figure 3: An almost-identical, but obviously *un*safe fix. A small syntactic mistake carries significant semantic weight: the `rm -fr` line can *only* delete everything user-writable.

Semantics of state transformations: A static analyzer must also be able to reason about how the aforementioned state is transformed by an arbitrary piece of shell code—*e.g.*, composition primitives, subshells, expansion, and built-ins.

Thus another ingredient is *symbolic execution*, simulating the actions of the shell interpreter, symbolically describing the results of operations and transforming sets of program states along the way. The symbolic engine implements the semantics of the shell language [6], including composition primitives such as `|`, `&`, and `&&`. It also models the behavior of key built-in commands, such as `cd` and `[`, analogously to primitive functions in other programming languages. During symbolic execution, the engine expands parameters, tracks working directories, follows success and failure paths—collecting and propagating constraints on symbolic variables and pruning via concrete state whenever possible.

For example, given that the possible contents of `$0` expand to either a directory or a filename, `cd` will either (1) change the program's current directory to a directory, print no output, and succeed, or (2) fail with output on `stderr` but not `stdout`. The engine will explore both executions and symbolically exit the subshell with two potential results. Upon conclusion, it will issue a warning for one of the two execution paths that results in `rm -fr /*`.

Inferring command specifications: The analysis depends on specifications of commands such as `cd` and `rm`, and their specific invocations such as `rm -fr`. Commands are fundamentally opaque, written by different developers and in arbitrary languages, often distributed as binaries, and equipped with a multitude of configuration flags and system-specific variations. Conventional program analysis techniques therefore face immense hurdles for inferring their specifications.

Fortunately, commands are typically distributed with some form of documentation. Thus, another ingredient is *documentation mining with instrumented probing*—applied ahead-of-time to build a queryable specification library accompanying the analysis engine. The first step (Fig. 4) is to derive a command's invocation syntax from its natural language documentation—*e.g.*, `man` pages, markdown files, web pages, *etc.* This syntax derivation leverages large-language models (LLMs) guardrailed via domain-specific languages designed to express only legitimate invocations—satisfying common utility conventions such as the XBD standard [13]. Using the syntax specification, a miner next generates a large number of test configurations sweeping through the possible flags, options, and relevant file system states. It then instantiates concrete environments, executing each command configuration with appropriate interposition to record all of its interactions within each environment. Finally, it examines the traces extracted by these executions and applies a series of transformation rules to produce the final specifications.

This process leads to a precise and accurate specification, expressed for example as Hoare triples—*i.e.*, preconditions and postconditions around the command invocation. For instance, for `rm -fr` the `man`-driven, LLM-assisted derivation generates a syntax specification that includes (1) `-r` and `-f` as distinct, non-exclusive flags, and (2) at least one positional argument to `rm` as a path. It then generates all valid invocations, including `rm { , -f, -r, -f -r} $p`, along with appropriate environments for probing each invocation's effects—including cases where `$p` is a file, a directory, or non-existent. By instrumenting its execution during probing, it discovers that given a path to an extant directory, `rm -f -r $p` deletes that directory and exits with code `0`:

$$\{(\exists\ \texttt{\$p}) \wedge (\texttt{arg 0 \$p path.FD})\}$$
`rm -f -r $p`
$$\{(\nexists\ \texttt{\$p}) \wedge \texttt{exit 0}\}$$

That is, *if* the `-r` and `-f` flags are present and the first positional argument is a path to a file or directory, *then* the file or directory is deleted and the exit code is 0.

Reasoning about stream contents: The analysis must also reason about the contents of Unix streams and files. To illustrate stream reasoning, consider Fig. 5's fix to Fig. 1's bug: append `STEAMROOT` with a platform-dependent subdirectory. The output of `lsb_release -a`, lines of label-value pairs separated by tabs, is filtered first by `grep` to select a line with a matching label, and then by `cut` to extract the second field. Unfortunately, `'^desc'` should have been `'^Desc'`: `grep` produces no output and the suffix is never set, leaving the script vulnerable to the same bug.

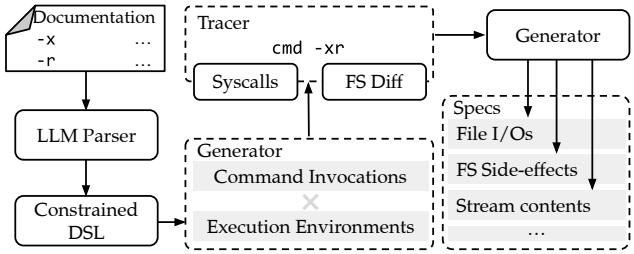

Figure 4: Command specification inference. Left: generate all valid invocations of a command by guardrailing a tuned LLM. Mid: instrument and execute all command invocations in a set of execution environments, confirming their effects. Right: compile their effects to specifications targeting key classes of constraints.

Encoding the shape of streams requires another ingredient: *regular types*, a new type system for string shapes centered around the familiar and concise representation of regular languages. The languages can equivalently describe the shape of entire streams or, more conveniently, of each line in the stream—*e.g.*, for the output of `lsb_release -a`:

```
(Distributor ID|Description|Release|Codename):\t.*
```

In turn, specifying that Fig. 5's `grep` accepts any input and produces output beginning with `desc` amounts to the following summary in the style of a type signature:

`grep '^desc'` :: `.*` → `desc.*`

Straightforward reasoning shows that the intersection of `grep`'s combined input (from `lsb_release`) and output constraints is the *empty* language, thus always resulting in the default (empty) `case`.

Key takeaways: Stepping back, the critical pieces to highlight about this entire discussion is that the proposed approach (1) performs the analysis entirely statically, well before the execution of a program, (2) stems directly from reasoning about the semantics of the shell and the components involved in a program, and (3) provides guarantees about all possible executions, not just a subset.

In contrast to other approaches (§2), this approach is context sensitive; it concludes safety or unsafety by tracking constraints on variable contents, including those from conditionals. It is also robust to semantically-equivalent syntactic variants such as splitting `rm`'s path across variables:

```
c="/*"; rm -fr $STEAMROOT$c
```

4 ADDITIONAL COMPLEXITY…

The four ingredients (§3) are potent enough to offer conclusive results on the snippets shown. Additional complexities may arise in practice.

File system effects: Diagnosing dangerous file deletions is an important benefit of the proposed analysis, but precise reasoning could detect and prevent even finer-grained file manipulation bugs (such as idempotence violations). By augmenting the symbolic execution engine, the analysis can track constraints on the nodes in the file system to which individual paths resolve; when competing constraints are inconsistent, the system determines that the script contains a bug arising from command composition.

As a simple yet suggestive example, consider the snippet on the right. By symbolically modeling the state of the file system, the analysis determines that, after the invocation of `rm`, the file or directory to which `$1` resolved no longer exists. The invocation of `cat` introduces a constraint that `$1/config` resolves to a file. To resolve this path, however, `$1` must resolve to a directory with a file entry named `config`, and the directory does not exist: the system issues a warning that the invocation will *always* fail.

```
rm -r "$1"
cat "$1/config"
```

While such bugs may seem trivial at the surface, they easily become subtle as multiple control paths, variables containing arbitrary paths, and path aliasing come into play. For example, understanding that Fig. 2's check on the normalized-path result of `realpath` implies information about the potentially un-normalized-path of `$STEAMROOT` demands reasoning about the identity of filesystem locations referrable to by arbitrarily many path-strings. And in general, file system bugs may be split across hundreds of lines of code and arise only under specific conditions. But this is all par for the course with a symbolic execution that simultaneously considers all possible executions. The central challenge is to track the file system's state with sufficient precision as to enable useful reasoning while avoiding exponential explosion in complexity for realistically sized programs.

Environment and runtime monitoring: Inference of stream and file contents, while feasible, still faces precision limitations, and therefore might at times fail to produce a type for a command. Addressing these limits, without risking safety and when user annotations are not available, requires runtime monitoring: when enabled, runtime monitoring protects computations adjacent to an untyped command to ensure their type expectations are maintained during the execution of the program.

This monitoring is achievable with a higher-order `monitor` command, similar in spirit to `strace` and `xargs` (but more sanely named), that monitors a command's input and output streams to ensure they conform to the specification inferred by adjacent commands. The cost of maintaining safety without annotations is monitoring overhead and delayed error detection—trade-offs similar to gradual types [11, 25, 32].

Ergonomic annotations: Retrofitting existing shell interpreters with an elaborate stream type system will likely require large amounts of engineering effort, drastic changes to syntax and semantics, and break backwards-compatibility.

```
#!/bin/sh
STEAMROOT="$(cd "${0%/*}" && echo $PWD)"/
case $(lsb_release -a | grep '^desc' | cut -f 2) in
  Debian) SUFFIX=".config/steam" ;;
  *Linux) SUFFIX=".steam" ;;
esac
rm -fr $STEAMROOT$SUFFIX
```

Figure 5: Yet another fix, introducing a subtle bug. The `grep '^desc'` filter is mistaken, allowing no content to pass through, causing the fix to never apply.

In order to avoid that and maintain full compatibility with existing shell interpreters, these constraints should instead join the shell ecosystem through annotations manifesting as *specialized inline comments or external files*. This strategy also allows developers to examine, interact, and produce new constraints as they see fit.

Confidence on non-expert annotations can be increased by levering instrumented probing (see §3, ingredient no.3), injecting runtime monitoring, or building a community-sourced repository of annotations à la TypeScript.

But dealing with raw constraints is intimidating and cumbersome. Alleviating this challenge requires offering first-class support for constraints, including an extensible library of descriptive types. For example, `any` may stand for `.*`; `url` for inputs to `curl`; and `longlist` for outputs of `ls -l`. Other support includes type introspection via shell utilities such as `typeOf`, type definitions and abstractions, and tight integration with dynamic monitoring infrastructure. Such support simplifies development, improves precision and performance, and results in more descriptive error messages.

Richer types: Simple regular types, as introduced above, do not always preserve information across stages. Consider the pipeline on the right, which extracts hexadecimal numbers, then prefixes every number with `0x`, and finally sorts them. Simple types for the first two stages are:

```
hex='[0-9a-f]+'
grep -oE "$hex" |
  sed 's/^/0x/'|
  sort -g
```

`grep -oE "$hex"` :: `.*` → `[0-9a-f]+`
`sed 's/^/0x/'` :: `.*` → `0x.*`

These two types alone are unable to establish that the input of `sort -g` falls under `0x[0-9a-f]+.*`, because the type of `sed` fails to propagate that each `0x` prefix is followed by a hexadecimal number.

Polymorphic types [21] come to the rescue, offering expressiveness that allows a single abstract type to be parameterized over one or more constituent types. For example, if every input line of `sed 's/^/0x/'` satisfies, say, α, then every output line satisfies `0x`α:

`sed 's/^/0x/'` :: $\forall\alpha.\ \alpha \to$ `0x`α

`sort -g` is similar, but imposes a constraint on α:

`sort -g` :: $\forall\alpha \subseteq$ `0x[0-9a-f]+.*` . $\alpha \to \alpha$

Correctness is confirmed by (1) instantiating `sed`'s type variable α with its concrete input `[0-9a-f]+` (from `grep`) to obtain the concrete output type `0x[0-9a-f]+`, and (2) confirming that this concrete output type is compatible with `sort -g`, *i.e.*, that `0x[0-9a-f]+` $\subseteq$ `0x[0-9a-f]+.*`.

Feedback loops and circular dataflow: As noted by Doug McIlroy on a recent HotOS panel on the Shell [7], complex patterns of interprocess communication in the shell are increasingly prevalent [10, §4.2]. Typical workloads such as crawlers and indexers already feature circular dataflow structures, but the rise of machine-learning workloads makes extensive use of feedback loops and backpropagation edges. These structures complicate type constraint propagation, requiring (challenging, in the worst case) fixpoint reasoning.

Establishing correctness in these settings requires reasoning about stream invariants, *i.e.*, properties preserved by the entire cycle. Worst-case invariant discovery for circular streams can be difficult, but real scripts require invariants simple enough to be computed with an iterative "least fixpoint" approach: start with an empty invariant set and then gradually expand it until a property needs no further expansion—often straightforward due to the semantics of `cat` or `tail -f` typically at the beginning of such cycles.

Incorrectness criteria: Deleting the entire file system, in the many ways triggered earlier, clearly constitutes buggy behavior. Other behaviors are not as clear, especially since the shell lacks well-established definitions of program correctness and is resilient to mistakes typically causing outright failure in other environments—*e.g.*, undefined variables or unhandled exceptional behavior.

A comprehensive and practically useful set of criteria requires surveying the literature and exploring bugs in the wild. For example, beyond dangerous destructive file system operations, the CoLiS project reveals idempotence as an important criterion for software installation scripts [15]. And an exploration of shell-related StackOverflow questions identifies many reasons for considering scripts buggy, ranging from command failures to messages on standard-error.

5 …AND BROADER HORIZONS

Building a robust infrastructure tackling these challenges for real shell programs in turn opens up exciting opportunities for future research across fields.

Correctness: Early detection of potential incorrectness is not limited to the script itself but can be extended to incompatibilities with the execution environment. For example, a script might be written on a developer environment running Mac OS X but might end up deployed on cloud infrastructure running Linux. Given the capability to perform static

reasoning, the analysis can trivially also identify and warn developers before distribution about platform-dependent code, and even automatically transform the program to equivalent variations for different platforms.

Besides identifying potential errors, static analysis can be leveraged to automatically insert fixes targeting correctness. These might include synthesized dependency prologues that ensure that a script's dependencies are met—including expected file system state, available utilities, and shell environment. With integration into IDE tooling, developers could also receive tailored suggestions that guarantee protection against identified bugs. Correctness benefits extend to other environments that operate as thin shell wrappers, such as Dockerfiles, Vagrant shell provisioners, GitHub Actions, *etc.*

Security: Infrastructure for ahead-of-time semantics-driven analysis can also offer significant security benefits [3, 22]. One possibility enabled by this infrastructure is checking scripts against user-defined specifications capturing properties beyond crashes or catastrophic system effects, which might even be context-dependent. Consider for example the unfortunately-not-uncommon `curl`-to-`sh` software installation method, which instructs users to download and execute an installer script with a single line of code:

```
curl sw.com/up.sh | sh
```

A security-conscious user may wish to confirm or enforce that `up.sh` does not read or modify key directories. With access to a configurable ahead-of-time specification `verify`er (and perhaps a library of commonly used definitions), they would prefer executing the following instead:

```
curl sw.com/up.sh | verify --no-RW ~/mine | sh
```

As the use suggests, such a verification tool could verify as much as possible statically, inform the user about possible effects, and then leverage the guard and monitor generation mentioned above to fill gaps or address reasoning challenges in static verification, protecting from unintended effects.

Performance: Whole-program performance optimizations for shell scripts show great promise in prior work. Such optimizations include shell script or command elision [1], parallelization [28, 34], fusion [12], and various forms of distribution [19, 23, 27, 35]. With rich static information, these optimizations can be improved by reducing the need to discover and reason about optimizations dynamically. As a concrete example, shell state and file system reasoning can identify read-write dependencies between commands in a script, which would allow speculative execution systems like hS [18] to reorder commands without needing to guard against misspeculation, and incremental execution systems like Riker [2] to reduce the runtime tracing overhead.

Alternatively, ahead of time analysis can be used to bring the benefits of shell optimization systems to users without asking them to replace their shell or change how they run scripts at all. A static optimization engine can serve as the backbone for a suggestion-based optimization coach that—similar to `ShellCheck`—can be integrated tightly with IDE tooling or even be available via a webpage.

Comprehension: Rich static information about shell script behavior opens up opportunities to help script developers, often experts in domains outside computer science, understand the behavior of their own code.

An interactive program visualization system, identifying possible behaviors and allowing users to explore the impact of different environments or assumption violations, could make all the difference for identifying or exploring desirable and undesirable behavior beyond strict incorrectness [24, 30, 31]. Such human-centered visualizations may still retain the shell's complex actions and effects while being interpretable even by developers with limited programming background.

. . . and beyond: The ongoing project to deliver correctness guarantees for computing systems has mainly focused on specific environments and languages. Yet, modern computing infrastructure remains heterogeneous and legacy systems are pervasive. The shell's continued prevalence is explained in great part by its role in *bridging the gap* between these heterogeneous systems—fundamentally, its key domain.

While the shell is not sacred, and many use-cases might benefit from replacing the shell with more tailored workflow languages, the fundamental challenges of bridging the gap are not particular to the language. The benefits to be found from seriously tackling correctness for the shell apply to *all* systems that bridge this gap, including *e.g.*, Docker, Ansible, CI/CD, and beyond. Heterogenous system orchestration, composition, and installation does not need to be a painful trouble spot that wastes developer time and causes mysterious bugs.

The series of insights outlined here hints at the feasibility of building tooling for the shell on par with that of other production languages and serves as a call for action.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. We also thank Brown CS2952R (Fall'24) participants for input on several iterations of this paper. This material is based upon research supported by NSF awards CNS-2247687, CNS-2312346, and CCF-2340479; NSF GRFP grant no. 2439559; DARPA contract no. HR001124C0486; a Fall'24 Amazon Research Award; a seed grant from Brown University's Data Science Institute; and a BrownCS Faculty Innovation Award.

REFERENCES

[1] Emery D Berger. 2003. *Optimizing Shell Scripting Languages.* Technical Report UMCS TR-2003-009. University of Massachusetts Amherst.

[2] Charlie Curtsinger and Daniel W Barowy. 2022. Riker: Always-Correct and fast incremental builds from simple specifications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 885–898. https://www.usenix.org/conference/atc22/presentation/curtsinger

[3] Ting Dai, Alexei Karve, Grzegorz Koper, and Sai Zeng. 2020. Automatically detecting risky scripts in infrastructure code. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 358–371. https://doi.org/10.1145/3419111.3421303

[4] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. 2017. NoFAQ: synthesizing command repairs from examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 582–592. https://doi.org/10.1145/3106237.3106241

[5] Vidar Holen et al. 2012. ShellCheck: A shell script static analysis tool. https://www.shellcheck.net/. Accessed: 2024-10-14.

[6] Michael Greenberg and Austin J. Blatt. 2019. Executable formal semantics for the POSIX shell. *Proc. ACM Program. Lang.* 4, POPL, Article 43 (2019), 30 pages. https://doi.org/10.1145/3371111

[7] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. 2021. The Future of the Shell: Unix and Beyond. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 240–241. https://doi.org/10.1145/3458336.3465296

[8] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. 2021. *HotOS 2021: Unix Shell Programming: The Next 50 Years (Fun Applications)*. ACM SIGOPS Youtube. https://www.youtube.com/watch?v=dMrfLCjtHM4#t=300s

[9] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. 2021. Presentation: Unix Shell Programming: The Next 50 Years. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 104–111. https://doi.org/10.1145/3458336.3465294

[10] Michael Greenberg, Konstantinos Kallas, Nikos Vasilakis, and Stephen Kell. 2021. Report on the "The Future of the Shell" Panel at HotOS 2021. arXiv:2109.11016 [cs.OS]

[11] Ben Greenman and Matthias Felleisen. 2018. A Spectrum of Type Soundness and Performance. *Proc. ACM Program. Lang.* 2, ICFP, Article 71 (2018), 32 pages. https://doi.org/10.1145/3235045

[12] Anna Herlihy, Periklis Chrysogelos, and Anastasia Ailamaki. 2022. Boosting efficiency of external pipelines by blurring application boundaries. In *12th Annual Conference on Innovative Data Systems Research (CIDR'22)*. Chaminade, CA, USA. https://infoscience.epfl.ch/entities/publication/79f2a485-ecb5-4272-a107-bf0951d5f6aak

[13] IEEE and The Open Group. 2018. *The Open Group Base Specifications Issue 7, 2018 edition. Volume XBD: Base Definitions.* https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/

[14] Github Inc. 2024. The top programming languages. https://github.blog/news-insights/octoverse/octoverse-2024/#the-most-popular-programming-languages.

[15] Nicolas Jeannerod. 2021. *Verification of Shell Scripts Performing File Hierarchy Transformations.* Ph. D. Dissertation. University of Paris.

[16] Idan Kamara. 2016. explainshell: match command-line arguments to their help text. https://explainshell.com/ Accessed: 2024-10-20.

[17] Shriram Krishnamurthi. 2003. *Programming Languages: Application and Interpretation* (3 ed.). https://www.plai.org/

[18] Georgios Liargkovas, Konstantinos Kallas, Michael Greenberg, and Nikos Vasilakis. 2023. Executing Shell Scripts in the Wrong Order, Correctly. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 103–109. https://doi.org/10.1145/3593856.3595891

[19] Aurèle Mahéo, Pierre Sutra, and Tristan Tarrant. 2021. The serverless shell. In *Proceedings of the 22nd International Middleware Conference: Industrial Track (Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 9–15. https://doi.org/10.1145/3491084.3491426

[20] Karl Mazurak and Steve Zdancewic. 2007. ABASH: finding bugs in bash scripts. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security (PLAS '07)*. Association for Computing Machinery, New York, NY, USA, 105–114. https://doi.org/10.1145/1255329.1255347

[21] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375.

[22] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. 2014. SHILL: a secure shell scripting language. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Broomfield, CO, USA, 183–199. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/moore

[23] Tammam Mustafa, Konstantinos Kallas, Pratyush Das, and Nikos Vasilakis. 2023. DiSh: Dynamic Shell-Script Distribution. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, USA, 341–356. https://www.usenix.org/conference/nsdi23/presentation/mustafa

[24] Brad A. Myers. 1990. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing* 1, 1 (1990), 97–123. https://doi.org/10.1016/S1045-926X(05)80036-9

[25] Max S. New, Daniel R. Licata, and Amal Ahmed. 2019. Gradual type theory. *Proc. ACM Program. Lang.* 3, POPL, Article 15 (2019), 31 pages. https://doi.org/10.1145/3290328

[26] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[27] Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia. 2020. POSH: A Data-Aware Shell. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 617–631. https://www.usenix.org/conference/atc20/presentation/raghavan

[28] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. 2022. Automatic Synthesis of Parallel Unix Commands and Pipelines with KumQuat. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 431–432. https://doi.org/10.1145/3503221.3508400

[29] Michael Sippel and Horst Schirmeier. 2023. Process Composition with Typed Unix Pipes. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems (PLOS '23)*. Association for Computing Machinery, New York, NY, USA, 34–40. https://doi.org/10.1145/3623759.3624546

[30] Juha Sorva. 2013. Notional machines and introductory programming education. *ACM Trans. Comput. Educ.* 13, 2, Article 8 (2013), 31 pages. https://doi.org/10.1145/2483710.2483713

[31] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Trans. Comput. Educ.* 13, 4, Article 15 (2013), 64 pages. https://doi.org/10.1145/2490822

[32] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of typed scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 395–406. https://doi.org/10.1145/1328438.1328486

[33] Github user `keyvin`. 2015. Moved ~/.local/share/steam. Ran steam. It deleted everything on system owned by user. #3671. https://github.com/ValveSoftware/steam-for-linux/issues/3671. Accessed: 2025-01-01.

[34] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. 2021. PaSh: Light-Touch Data-Parallel Shell Processing. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 49–66. https://doi.org/10.1145/3447786.3456228

[35] Keith Winstein and Hari Balakrishnan. 2012. Mosh: an interactive remote shell for mobile clients. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, USA, 15. https://www.usenix.org/conference/atc12/technical-sessions/presentation/winstein