Unix Shell Programming: The Next 50 Years

Michael Greenberg*
Pomona College
Claremont, CA, USA
michael.greenberg@pomona.edu

Konstantinos Kallas*
University of Pennsylvania
Philadelphia, PA, USA
kallas@seas.upenn.edu

Nikos Vasilakis*
MIT
Cambridge, MA, USA
nikos@vasilak.is

ABSTRACT

The Unix shell is a powerful, ubiquitous, and reviled tool for managing computer systems. The shell has been largely ignored by academia and industry. While many replacement shells have been proposed, the Unix shell persists. Two recent threads of formal and practical research on the shell enable new approaches. We can help manage the shell's essential shortcomings (dynamism, power, and abstruseness) and address its inessential ones. Improving the shell holds much promise for development, ops, and data processing.

CCS CONCEPTS

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

KEYWORDS

Shell, Unix, JIT, Analysis, Transformation, Optimization

ACM Reference Format:
Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. 2021. Unix Shell Programming: The Next 50 Years. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31–June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458336.3465294

1 INTRODUCTION

The Unix shell is an attractive choice for specifying succinct but powerful scripts for system orchestration, data processing, and other automation tasks. Moreover, it is the first tool that one has access to in every Unix distribution. Its benefits and its unavoidability make the shell extremely well exercised, if not well loved, in the current status quo. While in principle cloud computing deprecates many shell tasks as automated systems take on various configuration and management jobs, in practice, shell scripts show up everywhere in the cloud regime: Docker, Vagrant, Kubernetes, and other cloud deployments are all managed by shell scripts.

There is no point denying it: the shell has many shortcomings and is hated by many. Shell scripts are unmaintainable, easy to get wrong, and inflexible with respect to performance; once a script is written it is difficult to refactor it to avoid redundant computation or properly utilize all the available computational resources. To make matters worse, the shell has been mostly left for dead by both academia and industry, considering it an unsalvageable piece of junk that needs to be replaced at the first opportunity.

1.1 Shouldn't we just replace the shell?

What if we just gave up on the shell? After all, moving from System V init scripts to systemd has arguably improved the Linux boot sequence. Why install packages manually when you can specify base images in Docker and Vagrant, and Travis CI will let you specify `addons` to provision per-platform. Moves to systemd, Docker, Vagrant, and other 'modern' services have indeed improved things a great deal, but there have been serious regressions. And, at core, these tools all use the shell extensively! Systemd uses its own variable expansion regime, slightly different from the shell's... encouraging you to run `sh -c` if you have an actual pipeline to run. Dockerfiles and CI config files are *almost* shell scripts, but there's no convenient way to just execute the `RUN` commands or `script` lists they contain. And few are the Vagrantfiles that don't call out to some `provision.sh` to set up dependencies. Giving up on the shell means that each 'modern' system tool will have its own janky quasi-shell language: a decidedly worse situation. Similarly, some simple shell scripts could just as well be Python scripts, but complex pipelines are much harder to port over. Using `popen` or even support libraries like Plumbum lack the easy composition enjoyed in the shell [23, 28].

1.2 Is it really that bad?

For a long time, the shell was neglected by both academia and industry as irredeemably bad. In our diagnosis, the desire to give up on the shell stems from three of its essential

*Alphabetical order.

HotOS '21, May 31–June 2, 2021, Ann Arbor, MI, USA

ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465294

characteristics that make it hard to think clearly about the shell and hard to reach its full potential:

(1) The shell contains a language that can be used to compose arbitrary commands, written in arbitrary languages and feature arbitrary behaviors.
(2) The shell's execution depends on a variety of dynamic components, such as the state of the file system and the values of environment variables.
(3) The shell's semantics is black magic, specified in a 119 page impenetrable document that is the POSIX shell specification (with an extra 160pp on utilities!).

These three characteristics stymie principled approaches to the shell, as any principled solution needs to (i) apply to a wide variety of arbitrary commands, (ii) account for its dynamic nature, and (iii) handle the intricacies of its semantics.

1.3 The shell is actually good

The shell is a useful abstraction deserving of our attention despite its imperfections (Section 2). Two recent projects—PaSh [49] and POSH [43]—gave us a glimpse of the possibilities, taming the aforementioned shell characteristics, transforming a restricted-but-widely-used fragment of the shell to data flow graphs. These graphs can be heavily optimized to produce fast, data-parallel interpretations of what were ordinary shell pipelines. We argue that by drawing from these efforts and recent work on mechanized shell semantics [24], we can rehabilitate the shell (Section 3).

The shell is a critical corner of computing; let it enjoy the advantages of nearly fifty years of systems and languages research that has neglected it. Better performance, safer expressivity, and gentler user experience await (Section 4).

2 THE GOOD, THE BAD, AND THE UGLY

The shell is a mixed bag: ubiquity, power, clumsiness, obscurity. Here are some aspects of the shell that are (i) worth keeping (*the good*), (ii) essential but challenging (*the bad*), and (iii) inessential flaws that can be addressed (*the ugly*).

2.1 The Good

The Unix shell hits a sweet spot between succinctness, expressive power, and performance. McIlroy presents a striking example [9], but the shell can go further: over 100 lines of Java code that perform a temperature analysis task [52] can be translated to a 48-character four-stage pipeline of comparable performance.

```
cut -c 89-92 | grep -v 999 | sort -rn | head -n1
```

So what are the ingredients of the shell's success?

G1: Universal composition. The shell is a natural composer of programs. More than any other language, the shell makes it easy to combine a variety of existing tools written themselves in a broad array of languages. Composing existing tools increases productivity, reliability, and simplicity—after all, "the most radical possible solution for constructing software is not to construct it at all" [12]. Unix helps by promoting a component philosophy [36] and by serving as the *de facto* component library [34], with most programs and libraries offering CLI frontends.

G2: Stream processing. The Unix shell embeds a small domain-specific language (DSL) for expressing pipelined stream computations—semantically close to Kahn process networks [30] and co-routines [31]. Coupled with filesystem-backed naming (of both code and data), language-agnostic composition, and higher-order primitives like `xargs`, the expressiveness of this DSL turns out to be quite powerful. Due to careful engineering of the pipe primitive and commands such as `sort` and `grep`, these streaming pipelines are capable of gracefully scaling to handle large amounts of data quite efficiently even if the underlying machine has limited resources.

G3: Unix-native. The features and abstractions of the shell are well suited to the Unix file system and file-based abstractions. Unix can be viewed as a naming service, mapping strings to longer strings, be it data files or programs. Naming is both convenient and essential to (powerful classes of) computation [46]. Processes, `PATH` entries, files, streams, and environment variables are all names, and the shell is the tool for manipulating and interacting with these names.

G4: Interactive. The shell is not just a programming language, but the lived-in environment. By having the entire environment be the program context, the shell lowers the barrier between interactive and non-interactive use. Interactivity is further facilitated by commands that are short and which often take single-letter flags with default options informed by real practical use [9].

2.2 The Bad

Some of the shell's essential characteristics make life difficult and prevent us from improving the shell. It's hard to imagine 'addressing' these characteristics without turning the shell into something it isn't; it's hard to get the good of the shell without these bad qualities [23].

B1: Too arbitrary. The shell's virtue of limitless composition (**G1**) is also its vice: the shell can compose arbitrary commands written in arbitrary languages. Any 'simple' shell command may translate to an `execve` of some arbitrary executable with arbitrary, unknown behaviors. Calls to `execve` make unified analysis very challenging, as different source languages won't share semantics; binary analysis—the lowest common denominator—cannot discover high-level invariants. This acts as a very high barrier of entry for research

and development of tools and analyses for the shell since researchers have to either make a huge effort to hardcode the semantics of each command for their tools to work, or risk the tool not being used widely.

B2: Too dynamic. Shell scripts are very succinct partly because of the existence of global state that can be modified and accessed at runtime. Unfortunately, this means that the behavior of a shell program cannot be known statically: a simple `grep $PWD -in ~/.*shrc` depends primarily on dynamically computed values, including the state of the file system, the current directory, environment variables, and unexpanded strings. This disallows any form of static analysis or transformation for improving the correctness and performance of shell scripts; a static analysis would either have to be unsound, assuming that the state ahead of time will be the same as the state at runtime, or ineffective, conservatively assuming the worst for every part of the state that can be modified at runtime.

B3: Too obscure. The semantics of the shell and common commands are documented in 300pp of standardese [7]. To be able to reason about a script's behavior, one needs to understand the exact behavior of its composition operators, the role of the environment, and the intricate state of the shell interpreter. Furthermore, there is no *single* shell environment. Multiple shells (with subtle behavior differences) coexist in the same machine: a pared down shell [4, 11] is used for startup scripts, while bash [45] is a common interactive choice. Every shell extends POSIX in its own way [24, 27]. The lack of an easy-to-use correctness baseline and the inability to formally reason about the shell's semantics inhibits research on the shell. On the other hand, developing new shells with cleaner semantics is likely to fail without any consideration for backward compatibility.

2.3 The Ugly

The bad aspects of the shell are part of its essence, making them hard to address without significantly diverging from the shell we know an love. In addition to them, the shell also has several flaws that (i) prevent it from being used for a wider variety of tasks, (ii) make the life of shell developers very difficult (leading to frustrated revulsion [19]), but (iii) aren't *essential* to its existence.

U1: Error-proneness. While all dynamic programming languages suffer from bugs that manifest as runtime errors, the shell is known for its many potential sources of error—with dire consequences! The shell's syntax and its direct access to the user's entire system, both aimed at terse interactive use, lead to a fast-paced high-stakes programming experience where a single typo could erase entire hard drives. Protective mechanisms such as assertions and error handling that are commonly used in critical code written in other languages are not well supported in the shell.

U2: Performance doesn't scale. While shell scripts have acceptable performance in a single-core setting, they're not tuned for multicore machines and clusters of nodes. Unlike other programming languages, the performance of shell scripts is dominated by the performance of the commands that they compose, and unfortunately, most shell commands do not scale. This leads users to restricted parallelism orchestration tools [20, 29, 48, 54] or even worse, to replace parts of their scripts with programs in parallel frameworks, an error-prone process that requires significant effort.

U3: Redundant recomputation. Small changes to the input of a script a complete re-execution, leading to many hours of wasted redundant computation. This is common in data processing (and preprocessing) workloads, as well as in build, configuration, and setup scripts. Domain specific solutions (such as build systems, *e.g.*, `make`) address this issue for their use cases, but do not generalize or compose.

U4: No support for contemporary deployments. The shell's core abstractions were designed to facilitate orchestration, management, and processing on a single machine. However, the overabundance of non-solutions—*e.g.*, `pssh`, GNU `parallel`, web interfaces—for these classes of computation on today's distributed environments indicates an impedance mismatch between what the shell provides and the needs of these environments. This mismatch is caused by shell programs being pervasively side-effectful, and exacerbated by classic single-system image issues, where configuration scripts, program and library paths, and environment variables are configured *ad hoc*. The composition primitives do not compose at scale.

3 ENABLERS

The shell has been around for half a century, so the addressable issues (**U1-4**) outlined in the previous section have been around, too. What's changed today? Three key enablers—two recent research threads and one proposed here—combine to unlock the shell's missing potential by alleviating the effects of the necessary evils (**B1-3**). These enablers have both conceptual merit, allowing others to build on their ideas, as well as practical value, as they are concrete open-source systems that can be used in other research.

3.1 Two Recent Enablers

Two recent enablers are (1) the formalization of the POSIX shell, and (2) light-touch parallelization and distribution transformation systems for the shell.

E1: Libdash & Smoosh Smoosh [24] is a recent effort to formalize the semantics of the POSIX shell, addressing the

obscurity and subtleties of the prose standard [7] (**B3**). It has two parts: (1) libdash, a linkable parsing library that supports both parsing POSIX shell scripts to ASTs and unparsing these ASTs back to scripts, and (2) a POSIX formalization mechanized in Lem, an ML-like language that can be compiled to Coq for mechanized reasoning and proofs or OCaml for execution. Smoosh has found several bugs in popular shell implementations, not to mention the POSIX specification and test suite itself.

Smoosh is a key enabler because it allows others to (1) study the POSIX shell and its nuanced behavior, (2) understand the divergences introduced by different shells, and (3) design and implement (formal) analyses and transformations on top of its symbolic, executable semantics.

E2: PaSh & POSH PaSh [49] and POSH [43] both proposed annotation languages as a high-level specification interface for dealing with the challenges of unknown command behavior (**B1**). Specifications are written once for each command and correspond to a specific command version (similarly to manpages). They can be aggregated in specification libraries which can be shared between users, not unlike completion libraries. PaSh and POSH use these specifications to transform shell pipelines to dataflow graphs [26] with specific properties, that they then optimize to either (1) achieve data parallelism in a multicore setting, speeding up slow CPU-intensive commands [49]; or (2) offload commands close to their input data, reducing network overhead [43].

PaSh and POSH specifications characterize important properties about commands—*e.g.*, their interaction with state and their inputs and outputs—and can be used as abstract models of the command behaviors and their interfaces, facilitating the development of analyses and transformations. A beneficial implication of the command specifications is that they decouple research on the shell from research on developing specifications for commands, allowing both to proceed independenly and in parallel, providing an interface for progress to be shared between them.

PaSh and POSH are key enablers because (1) they show that the shell too can benefit from automated transformation frameworks, (2) they offer a baseline for other tools to improve performance performance, (3) they propose annotation languages that enable reasoning about arbitrary commands, and (4) they propose a dataflow model for a subset of the shell—a foundation for further analyses and transformations.

3.2 A proposal...

PaSh and POSH showed that shell scripts can enjoy *order-of-magnitude* performance improvements with adroit preprocessing [43, 49]. But both systems face two serious impediments: the shell's dynamism and system resource constraints. Any attempt to reason statically about the shell will face these challenges. We propose a dynamically triggered optimization regime for the shell that we call Jash, short for 'Just a shell'. Jash inspects each shell command as it comes in to identify candidates for rewriting. Since Jash works dynamically, it can take into account current system conditions to decide whether to even try to apply optimizations!

The dynamic nature of the shell As discussed earlier (**B2**), the execution of shell scripts depends on several dynamic components such as the state of the file system, the current working directory, and the values of environment variables. Therefore, ahead-of-time solutions, such as the ones proposed by PaSh and POSH, choose between being conservative and ineffective or optimistic and unsound. Consider the following script, which is based on the original `spell` program by Johnson [8], lightly modified for modern environments.

```
FILES="$@"
cat $FILES | tr A-Z a-z |
    tr -cs A-Za-z '\n' | sort -u | comm -13 $DICT -
```

The above script checks the spelling of a set of input `$FILES` based on a dictionary file that is referenced by the environment variable `$DICT`. An ahead-of-time compiler has no knowledge of the input files and thus cannot properly decide if and how to parallelize or distribute the above pipeline—*i.e.*, neither PaSh nor POSH optimize this script.

To support the shell's dynamism (**B2**) Jash uses a just-in-time (JIT)[1] compilation framework that invokes the compiler as late as possible to provide it with necessary runtime information. The JIT tightly couples with the shell, switching back and forth between interpretation and optimization; interpretation is provided by the user's original shell and deals with dynamic features such as parameter expansion, and optimization is provided by the core analysis and transformation infrastructure and deals with optimization (for us, at first, parallelization à la PaSh). This line-oriented, shell-native architecture allows Jash to call the compiler at the right time—with as much runtime information available as possible—resulting in a system that can perform sound optimizations in both programmatic and interactive contexts.

By running just-in-time, the optimization subsystem has access to crucial information regarding performance optimizations, *e.g.*, file sizes, mappings from filesystems to physical media, and system load. Jash can determine in the moment whether it is even worth trying to optimize on small inputs. For the above example, the JIT compiler would first determine the input files, expand `$DICT`, and then determine how to parallelize the pipeline in lines 2-4. Expanding the

[1] Our definition of JIT is different from its traditional usage where JIT implies aggressive optimizations on hot parts of the code using profiling information. In our context, JIT simply means that the compiler is invoked at the right time with adequate information about the state of the shell and its environment—think "dynamic analysis for runtime optimization".

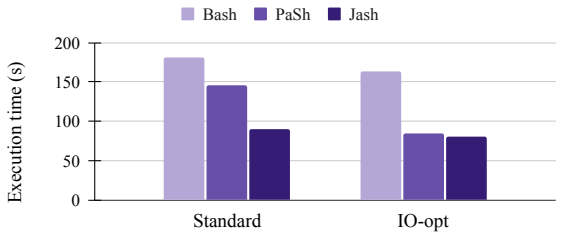

Figure 1: Executing a script that sorts the words of a 3GB input file with bash, PaSh, and the Jash prototype. Both instances are `c5.2xlarge` AWS EC2. The standard instance has a `gp2` disk (100 IOPS that bursts to 3K) while the IO-opt has a `gp3` disk (15K IOPS). PaSh performs worse on 'Standard' because it doesn't take system resources into account.

parameters before running the pipeline must be done with care: early expansions shouldn't have side-effects; Smoosh's semantics is critical for this kind of reasoning. To sum up: Jash will not only be sound with respect to shell semantics, it will also be more effective.

Restrictive Resource Assumptions PaSh and POSH focus on achieving performance improvements given an abundance of underlying computational resources. PaSh assumes a machine with high storage throughput and lots of available storage space for buffering. POSH assumes a cluster where computational resources per node are unlimited and thus co-locating the computation with the data never degrades performance. These assumptions simplify optimizations, but do not represent the entire population of shell users, which ranges from owners of palm-sized computers to administrators of supercomputers.

To relax these assumptions, we are developing a resource-aware optimization procedure that ensures performance improvements on a multitude of underlying platforms. The procedure is built on top of a cost-aware dataflow model, allowing for an extensible graph rewriting system that applies transformations with certain performance objectives within a specified cost budget. The JIT compiler keeps the optimization procedure up-to-date on the currently available resources of the underlying infrastructure as well as the size and characteristics of the input. Figure 1 shows the execution time of a script with bash, PaSh, and the Jash prototype on two EC2 instances with different IO capabilities—Jash exhibits better performance in both setting due to resource awareness.

The JIT compiler and resource-aware optimization yield a shell that can be used by *anyone* on *any infrastructure* and still lead to performance benefits (and no regressions!) for a wider variety of scripts and input workloads.

3.3 ...and Enabler

Jash includes a few other standalone contributions such as high-performance libdash-JIT interactions, but its primary contribution is its architecture, which enables future work.

E3: Jash Jash promises to operate on any POSIX shell script that exhibits the full range of dynamic behavior allowed by the standard—including but not limited to variable expansion, command substitution, process substitution. Jash's JIT-based approach helps us surmount an obstacle essential to the shell—namely, the limitations of static insights in the dynamic world of the shell (**B2**). Combined, **E1–3** open exciting new research directions for which dynamic interposition is the key tool.

4 THE FUTURE OF THE SHELL

The aforementioned enablers open exciting new directions in distributed computing, incremental computation, expressiveness support, and tooling for the shell.

Distribution Building a distributed Unix equivalent, in which Unix abstractions transcend single-computer boundaries, has been a goal since the 1970s [38, 40, 41, 50, 51]. Most attempts implemented full-fledged distributed operating systems, but enablers **E1–3** hint at the option of a language-systems hybrid: a thin but sophisticated rewriting-based shim à la Jash would have simplified the design of these systems, and would have armed them with the semantic foundation necessary for tackling unavoidable distribution trade-offs [5, 21, 35].

A few other connections can enable worthwhile research directions. PaSh and POSH identify a fragment of the shell with simpler semantics than the complete shell, *i.e.*, dataflow programs that take a set of inputs and produce a set of output files. This fragment can be easily manipulated; combining programs in this fragment with the JIT compilation of Jash and recent developments on the interactivity of remote shells [53] could enable the development of a well-behaved distributed and fault tolerant shell, where users can easily configure and efficiently execute tasks on a cluster of nodes. These insights could also be combined with recent work on developing and compiling applications on top of emerging serverless platforms, *e.g.*, gg [18] and Lambada [39], to explore porting the shell to this setting. For a subset of commands the shell can be thought of as a workflow description language; composing commands to configure a system or perform a complex task. Building on recent work [13] it would be possible to develop a fault tolerant and efficient shell implementation for workflows in cloud deployments. Finally, a deep understanding of the shell language (**E1**) can guide the design of distribution-friendly language subsets—with promising candidates being a streaming DSL [47], a

concurrency DSL [22], or a declarative configuration management DSL that could provide semantic foundations for systems such as Ansible [1], Chef [2], and Puppet [3].

Incremental Computation The command specification frameworks proposed by PaSh and POSH and the JIT optimization subsystem in Jash (**E2**, **E3**), can also serve as the foundation of an incremental computation framework to address **U3**. Incremental computation is especially useful in the context of the shell: developing a shell script is an iterative cycle of coding, testing, inspecting, and tweaking. Furthermore, shell scripts are often used for data downloading, extracting, cleaning, and other processing tasks. Re-executing such scripts on large datasets can be prohibitively expensive!

Incremental computation [44] has been widely studied for a variety of domains, from restricted ones, such as stream processing [37]; to general ones, such as threading support [10]. However, in almost all cases it requires extending the programming model in order to expose necessary information, *e.g.*, input-output dependencies that can be exploited by the IC compiler and runtime [14, 25]. Simply extending the shell language would break legacy scripts and restrict the possible adoption of such a framework.

PaSh and POSH's command specifications are the missing link, exposing the necessary information for an incremental computation framework. For example a command that processes each of its input lines independently need not be reapplied to the input lines that were unchanged. The JIT framework can then be used to provides up-to-date information on the latest state of script inputs. Combined, we have the critical building blocks for a runtime that incrementally reinterprets a script given changes of its input.

Heuristic support Perhaps the hardest part of supporting shell programming is reasoning about the commands users run. Both PaSh and POSH use command specifications to specify parallelization-relevant aspects of commands' behavior. These specifications were hand-written after carefully inspecting each command to determine its parallelizability properties. Hand-writing specifications even for common commands is quite a task, but users will need help not just with standard utilities, but their own programs (**B1**). Formal methods techniques such as fuzz testing, program analysis, and active learning could (i) test that a command conforms to its specification or even (ii) learn important aspects of a command's specification by inspecting its behavior. Testing and inferring command specifications would enable large libraries (formal, symbolic man pages) concerning a variety of properties (*e.g.*, concurrency/parallelism, expected command line argument syntax, filesystem access, expected runtime/memory usage).

Extending specifications with more properties would allow for a variety of expressive analyses—extending the syntactic checks of ShellCheck [33] and SecureCode [15], the man-page-directed listings of ExplainShell [32], or the purely textual model of NoFAQ [16]. Two promising directions: identifying errors and command misuse in a shell script, and a shell tutor (**B3**, **U1**). Building on the JIT execution framework and the command specification libraries, one could develop a sound JIT analysis that detects command misuse at runtime (but still before it occurs). Such an analysis could be based on expert written rules [15, 33] or even statistical data-driven techniques [6, 17, 42]. The tutor could use the library of specifications as a database to either answer queries about particular commands or to guide users while they develop a script.

User support The shell's user interface shows its age. The field of HCI seems to neglect the terminal as a relic, but active research on notebooks holds promise, and innovations in this domain could port over. Many projects focus on better interactive shells, at some cost to programmability [23]. Innovation in terminal emulators (like Fig and iTerm2) improve user experience, too. The historical divide between terminal and shell may no longer be appropriate: for example, iTerm2 uses a custom protocol of escape codes to munge `PS1` and detect prompts. More modern support—like language servers and structured protocols—will allow better, tighter integrations between the shell and the terminal. The online approach proposed in Jash is a natural way to achieve these integrations.

Formal support Smoosh provides formal support for building support systems for the shell (**B1**). Rich command annotations feed back into more robust symbolic execution and program analysis tools. The JIT optimization subsystem can be proved correct with respect to Smoosh's formal semantics and command annotations. Just as formal work on undefined behavior in C has supported both tool authors and standards writers, formal work on the shell promises continued improvements in standards and shared understanding.

5 REHABILITATING THE SHELL

Building on recent advances, the shell is a promising area of research. Neglected for so long, improving the shell will improve the experience for users of many stripes (development, ops, data processing, and novices). The shell warrants a fresh look from the research community.

Acknowledgments

We want to thank the HotOS reviewers, as well as Benjamin Pierce, Deepti Raghavan, Diomidis Spinellis, Jiasi Shen, Julia Lawall, Jürgen Cito, and Michael Schröder. This material is based upon work supported by DARPA contract no.

HR00112020013 and no. HR001120C0191, and NSF award CCF 1763514. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of DARPA or NSF.

REFERENCES

[1] 2020. Ansible is Simple IT Automation. https://www.ansible.com/

[2] 2020. Chef: Deploy new code faster and more frequent. https://www.chef.io/

[3] 2020. Puppet: Powerful infrastructure automation and delive. https://puppet.com/

[4] 2021. *Dash (Debian Almquist shell).* https://git.kernel.org/pub/scm/utils/dash/dash.git/

[5] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (2012), 37–42.

[6] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJOFETxR-

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

[8] Jon Bentley. 1985. Programming Pearls: A Spelling Checker. *Commun. ACM* 28, 5 (May 1985), 456–462. https://doi.org/10.1145/3532.315102

[9] Jon Bentley, Don Knuth, and Doug McIlroy. 1986. Programming Pearls: A Literate Program. *Commun. ACM* 29, 6 (June 1986), 471–483. https://doi.org/10.1145/5948.315654

[10] Pramod Bhatotia, Pedro Fonseca, Umut A Acar, Björn B Brandenburg, and Rodrigo Rodrigues. 2015. iThreads: A threading library for parallel incremental computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 645–659.

[11] Stephen R Bourne. 1978. *An introduction to the UNIX shell.* Bell Laboratories. Computing Science.

[12] Frederick P. Brooks, Jr. 1987. No Silver Bullet—Essence and Accidents of Software Engineering. *Computer* 20, 4 (April 1987), 10–19. https://doi.org/10.1109/MC.1987.1663532

[13] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. 2021. Serverless Workflows with Durable Functions and Netherite. *arXiv preprint arXiv:2103.00033* (2021).

[14] Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. 2011. Two for the Price of One: A Model for Parallel and Incremental Computation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 427–444. https://doi.org/10.1145/2048066.2048101

[15] 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

[16] 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

[17] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. HOPPITY: LEARNING GRAPH TRANSFORMATIONS TO DETECT AND FIX BUGS IN PROGRAMS. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SJeqs6EFvB

[18] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 475–488.

[19] Simson Garfinkle, Daniel Weise, and Steven Strassmann. 1994. *UNIX-Hater Handbook.* IDG Books Worldwide, Inc.

[20] Wolfgang Gentzsch. 2001. Sun grid engine: Towards creating a compute power grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE, 35–36.

[21] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 33, 2 (2002), 51–59.

[22] Michael Greenberg. 2018. The POSIX shell is an interactive DSL for concurrency. https://cs.pomona.edu/~michael/papers/dsldi2018.pdf.

[23] Michael Greenberg. 2018. Word Expansion Supports POSIX Shell Interactivity. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming* (Nice, France) *(Programming'18 Companion)*. Association for Computing Machinery, New York, NY, USA, 153–160. https://doi.org/10.1145/3191697.3214336

[24] 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 (Jan. 2020), 31 pages. https://doi.org/10.1145/3371111

[25] Matthew A Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S Foster. 2014. Adapton: Composable, demand-driven incremental computation. *ACM SIGPLAN Notices* 49, 6 (2014), 156–166.

[26] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin Rinard. 2020. An Order-aware Dataflow Model for Extracting Shell Script Parallelism. *arXiv preprint arXiv:2012.15422* (2020).

[27] Helmut Herold. 1999. *Linux-Unix-Shells: Bourne-Shell, Korn-Shell, C-Shell, bash, tcsh.* Pearson Deutschland GmbH.

[28] https://github.com/tomerfiliba/plumbum/graphs/contributors. 2018. *Plumbum: shell combinators.* http://plumbum.readthedocs.io/en/latest/

[29] Lluis Batlle i Rossell. 2016. *tsp(1) Linux User's Manual.* https://vicerveza.homeunix.net/ viric/soft/ts/.

[30] Gilles Kahn. 1974. The Semantics of a Simple Language for Parallel Programming. *Information Processing* 74 (1974), 471–475.

[31] Gilles Kahn and David B. MacQueen. 1977. Coroutines and Networks of Parallel Processes. *Information Processing* 77 (1977), 993–998.

[32] Idan Kamara. 2016. *explainshell.* http://explainshell.com/

[33] koalaman. 2016. *ShellCheck.* https://github.com/koalaman/shellcheck/

[34] Butler W Lampson. 2004. Software components: Only the giants survive. In *Computer Systems: Theory, Technology, and Applications*. Springer, 137–146.

[35] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The SNOW Theorem and Latency-Optimal Read-Only Transactions.. In *OSDI*. 135–150.

[36] M McIlroy, EN Pinson, and BA Tague. 1978. UNIX Time-Sharing System. *The Bell system technical journal* 57, 6 (1978), 1899–1904.

[37] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow.. In *CIDR*.

[38] Sape J Mullender, Guido Van Rossum, AS Tanenbaum, Robbert Van Renesse, and Hans Van Staveren. 1990. Amoeba: A distributed operating system for the 1990s. *Computer* 23, 5 (1990), 44–53. https://www.cs.cornell.edu/home/rvr/papers/Amoeba1990s.pdf

[39] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 115–130.

[40] John K Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. 1988. The Sprite network operating system. *Computer* 21, 2 (1988), 23–36. http://www.research.ibm.com/people/f/fdouglis/papers/sprite.pdf

[41] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. 1990. Plan 9 from Bell Labs. In *Proceedings of the summer 1990 UKUUG Conference*. 1–9. http://css.csail.mit.edu/6.824/2014/papers/plan9.pdf

[42] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[43] 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

[44] Ganesan Ramalingam and Thomas Reps. 1993. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 502–510.

[45] Chet Ramey. 1994. Bash, the Bourne-Again Shell. In *Proceedings of The Romanian Open Systems Conference & Exhibition (ROSE 1994), The Romanian UNIX User's Group (GURU)*. 3–5.

[46] Olin Shivers. 2018. *What's in a name?* https://www.ccs.neu.edu/home/shivers/papers/whats-in-a-name.html Accessed: 2018-09-27.

[47] Diomidis Spinellis and Marios Fragkoulis. 2017. Extending unix pipelines to dags. *IEEE Trans. Comput.* 66, 9 (2017), 1547–1561.

[48] Ole Tange. 2011. GNU Parallel—The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (Feb 2011), 42–47. https://doi.org/10.5281/zenodo.16303

[49] 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* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 49–66. https://doi.org/10.1145/3447786.3456228

[50] Nikos Vasilakis, Ben Karel, and Jonathan M. Smith. 2015. From Lone Dwarfs to Giant Superclusters: Rethinking Operating System Abstractions for the Cloud. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Switzerland) *(HOTOS'15)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=2831090.2831105

[51] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. 1983. The LOCUS distributed operating system. In *ACM SIGOPS Operating Systems Review*, Vol. 17. Acm, 49–70. http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.344

[52] Tom White. 2015. *Hadoop: The Definitive Guide* (4th ed.). O'Reilly Media, Inc.

[53] Keith Winstein and Hari Balakrishnan. 2012. Mosh: An interactive remote shell for mobile clients. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 177–182.

[54] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 44–60.