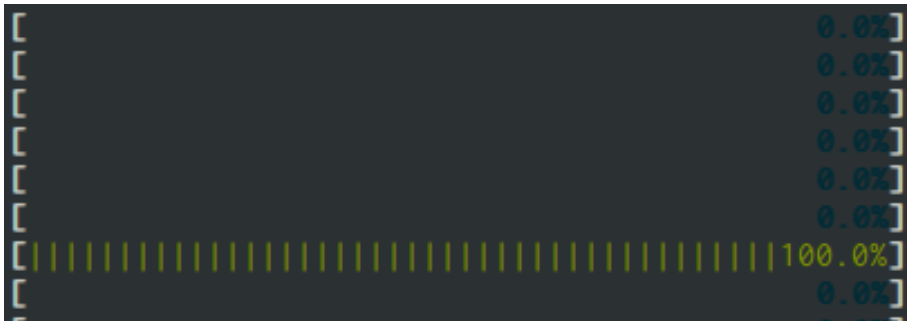


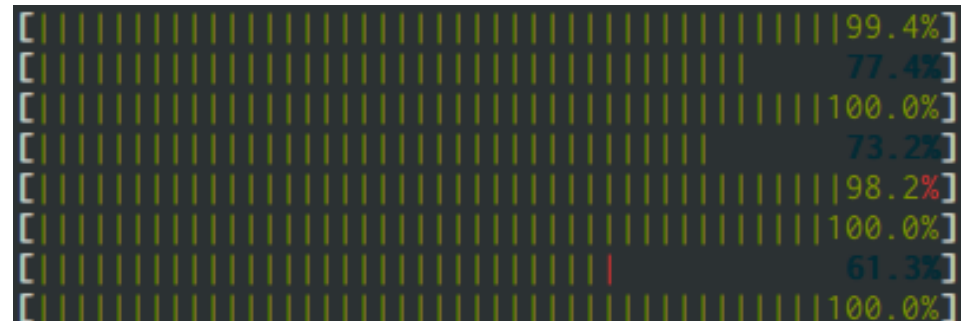
Practically Correct, Just-in-Time Shell Script Parallelization

Portland State University -- Spring 2023

or how to get from this:



to this:



Joint work with:



Tammam Mustafa
p: MIT now: Google



Jan Bielak
Staszic High



Dimitris Karnikis
p: Aarno Labs



Thurston Dang
p: MIT now: Google



Michael Greenberg
Stevens



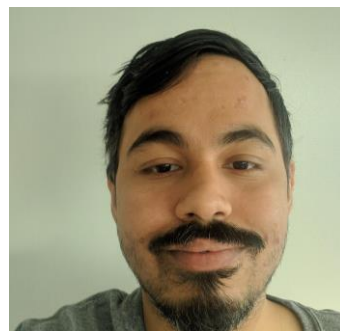
Nikos Vasilakis
Brown



Achilles Benetopoulos
UCSC



Lazar Cvetkovic
ETH



Shivam Handa
MIT



Kostas Mamouras
Rice



Radha Patel
p: MIT



Martin Rinard
MIT

shell

Used

```
#!/usr/bin/...
# TODO
# w
P

... loop as the one in execute_evaluation_scripts
microbenchmark=${flags[@]}
echo "Executing test: $microbenchmark"
# Execute the sequential script on the first run only
exec_seq="-s"
for n_in in "${n_inputs[@]}"; do
    echo "Number of inputs: ${n_in}"
    ## Generate the intermediary script
    python3 generate_microbenchmark_intermediary_script \
        $microbenchmarks_dir $microbenchmark $n_in $intermediary_dir "test"
    for flag in "${flags[@]:1}"; do
        echo "Flag: ${flag}"
        ## Execute the intermediary script
        ./execute_evaluation_script \
            "${microbenchmark}" "${n_in}" "test_results" "test_" > /dev/null 2>&1
        rm -f /tmp/eager*
    done
done
exec_seq=""

## Only run the sequential the first time around
execute_tests "" "${script_microbenchmarks[@]}"
execute_tests "-c" "${pipeline_microbenchmarks[@]}"
echo "Below follow the identical outputs:"
grep --files-with-match "are identical" ../evaluation/results/test_result

echo "Below follow the non-identical outputs:"
grep -L "are identical" ../evaluation/results/test_result

TOTAL_TESTS=$(ls -la ../evaluation/results/.. | wc -l)
PASSED_TESTS=$(grep --files-with-match "are identical" ../evaluation/results/test_result | wc -l)
echo "Summary: ${PASSED_TESTS}/${TOTAL_TESTS} tests passed"
```

id}'

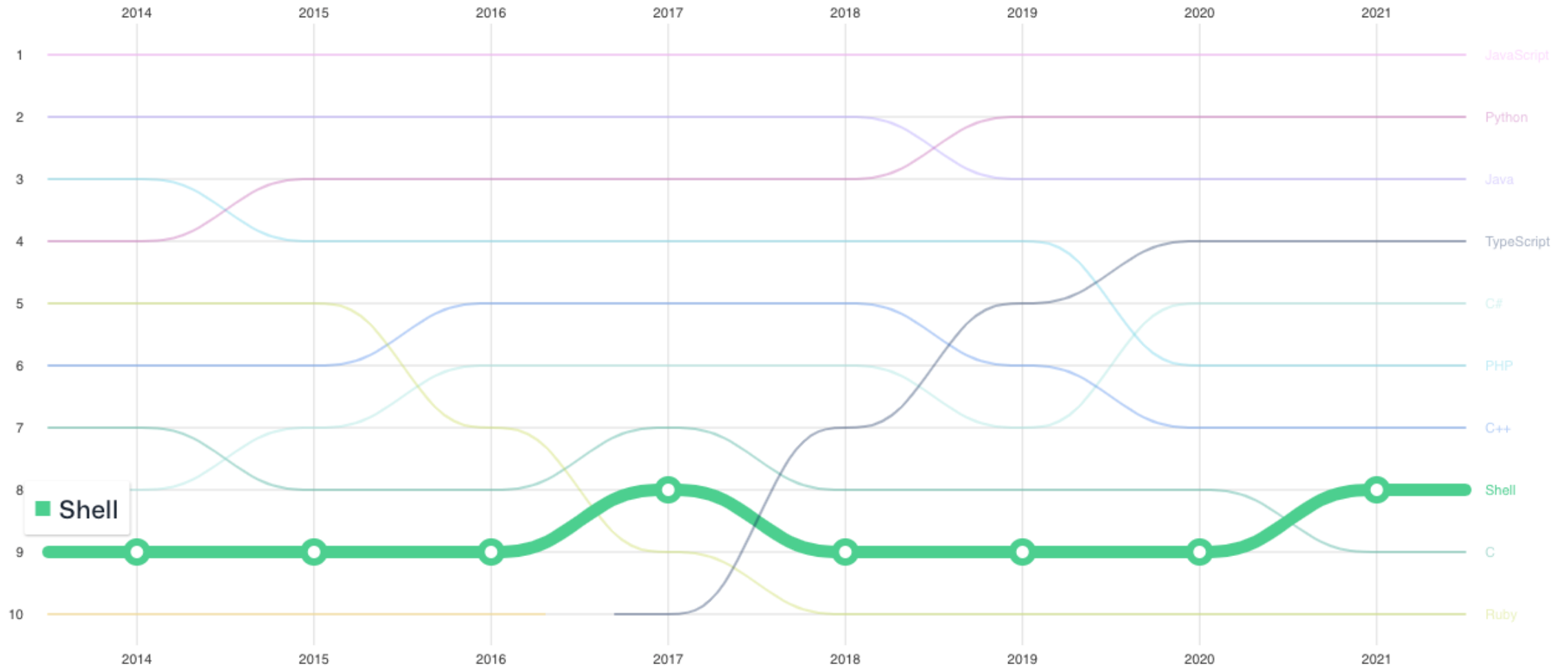
ver}')

.."

.."

.."

... for real



from the 2021 state of the octoverse: <https://octoverse.github.com>

Why? ... well, the shell is great

- Universal Composition
 - Composing arbitrary commands using files and pipes
 - Allows users to create powerful but succinct scripts
- Unix native
 - It is well suited to the Unix abstractions (files, strings, etc)
 - Offers great control and management of the file system
- Interactive
 - The complete system environment is accessible
 - Short commands and flags allows for quick experimentation

An example: Temperature Analysis

- This script computes the max temp in the US for the years 2015-2019

- To do so it:

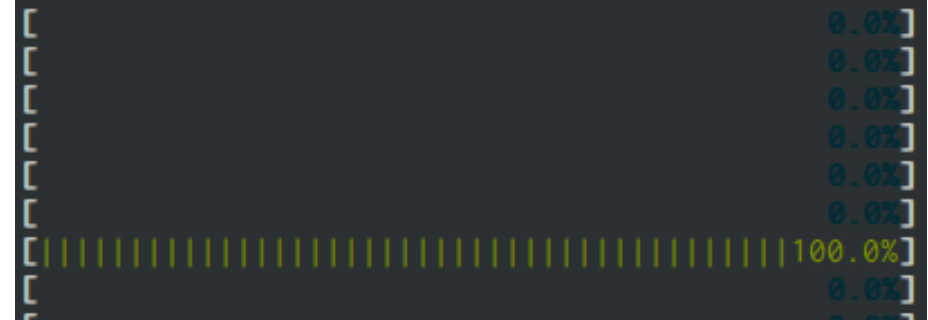
- Preprocessing {
 - Fetches the indexes of temperature data archives
 - Downloads the archived temp data
 - Extracts the raw data
- Processing {
 - Cleans it
 - Computes the maximum

```
base="ftp://ftp.ncdc.noaa.gov/pub/data/noaa";
for y in {2015..2019}; do
  curl $base/$y | grep gz | tr -s" " | cut -d" " -f9 |
  sed "s;^;$base/$y/;" | xargs -n 1 curl -s | gunzip |
  cut -c 89-92 | grep -iv 999 | sort -rn | head -n 1 |
  sed "s/^/Maximum temperature for $y is: /"
done
```

- The preprocessing part is taken from the Hadoop book
 - Until the gunzip
- The final two lines replace the MapReduce program from Hadoop book
 - The MapReduce equivalent in Java is 150 lines of code :')

The shell is great but ...

Shell scripts are mostly sequential!*



Parallelizing requires a lot of manual effort:

- Using specific command flags (e.g., `sort -p`, `make -jN`)
- Using parallelization tools (e.g., GNU parallel)
- Rewriting script in parallel languages (e.g. Erlang)

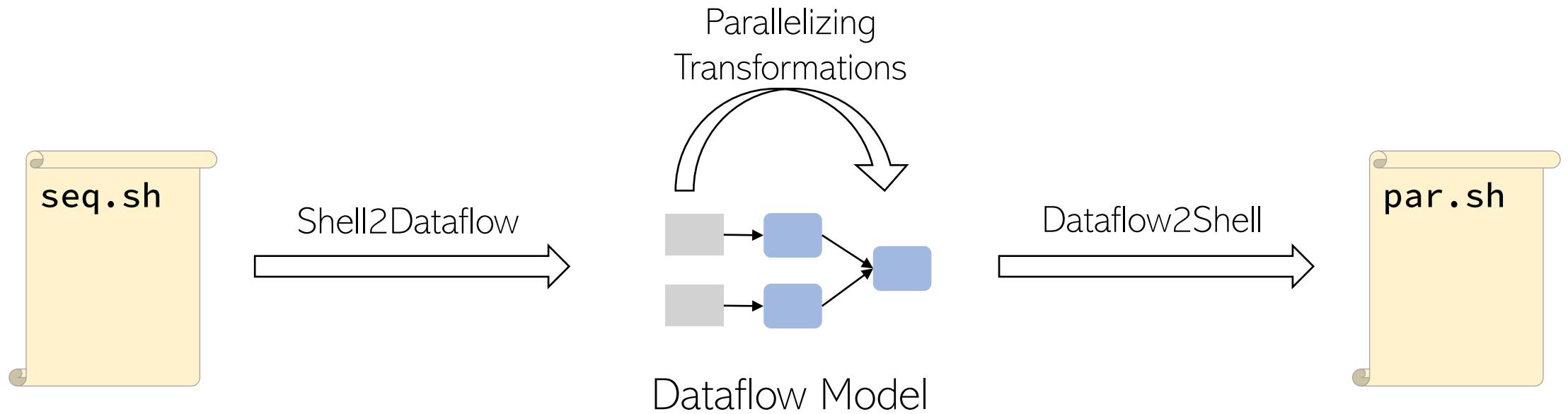


What did we do to deserve this??? :(

*Actually they have a ton more issues but we will come to that in the end

PaSh

PaSh



No tight coupling: Could work on top of any shell!

PaSh on Temperature Analysis

82GB (5y weather data)

Hadoop only focuses on this part

Preprocessing

Processing

bash

33m58s

10m4s

pash -w 16

16m39s

49s

2.04x

speedup for
preprocessing

12.31x

speedup for
processing

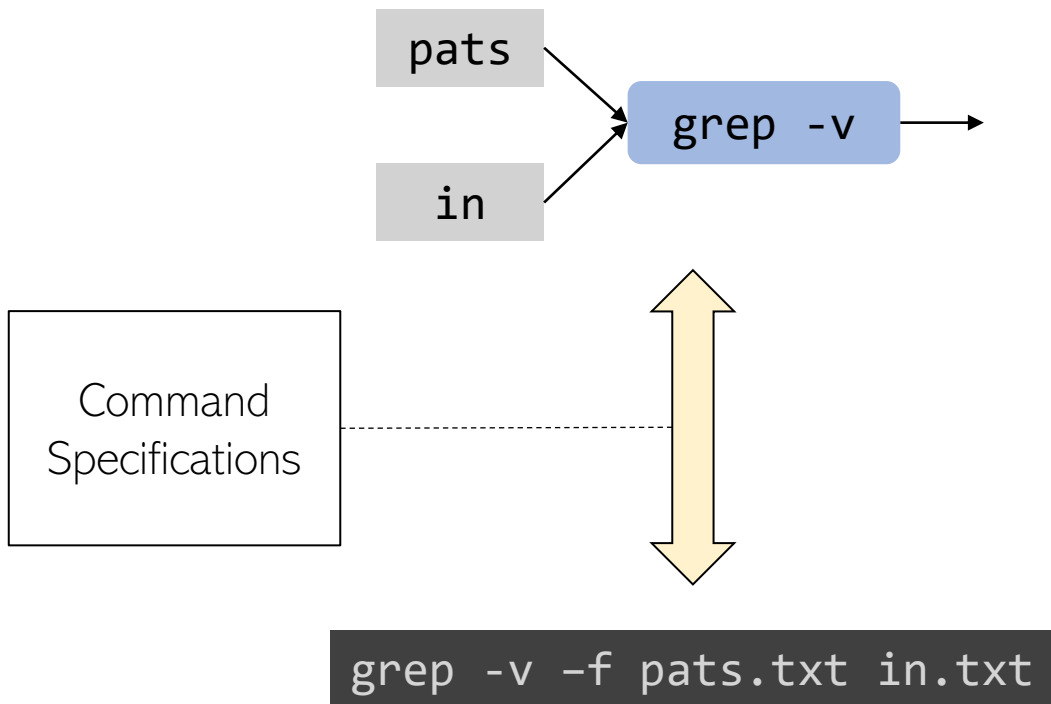
2.52x

combined speedup
for the full program

This part is not the focus of traditional parallelization frameworks but parallelizing it has the biggest impact

PaSh Insights

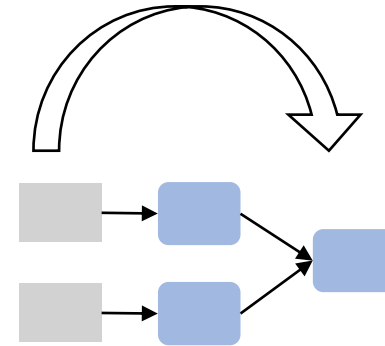
Command Specification Framework



Formalized 

Order Aware Dataflow Model

Parallelizing Transformations



 Transformations proven correct 

Read the PaSh papers at EuroSys 21 and ICFP 21 for more!

PaSh -- The static way

You should
do a static
analysis!



You should
do a static
analysis!



You should
do a static
analysis!



You should
do a static
analysis!



You should
do a static
analysis!



That should be OK, right?



Conservative or unsound – Choose one

- The shell is dynamic:
 - Current directory
 - Environment variables
 - Unexpanded strings
 - File system
- Static parallelization has to choose:
 - Sound but **conservative**
 - **Unsound** and optimistic

```
IN=${IN:-$TOP/pg}
mkdir $IN
cd $IN
echo 'Downloading, be patient...'
wget $SOURCE/data/pg.tar.xz
if [ $? -ne 0 ]; then
    echo "Download failed!"
    exit 1
fi
cat pg.tar.xz | tar -xJ

cd $TOP
OUT=${OUT:-$TOP/output}
mkdir -p "$OUT"
for input in $(ls ${IN}); do
    cat "$IN/$input" |
    tr -sc '[A-Z][a-z]' '[\012*]' |
    sort > "${OUT}/${input}.out"
done
```

PaSh-JIT

Just in time parallelization

- PaSh-JIT tries to parallelize as-late-as-possible™
- Provides critical information to the compiler:
 - State of shell, Variables, Directory, Files
- Not only correct, but also **faster!!!**

Just-in-time? How?

- By constantly switching between evaluation and parallelization
- PaSh-JIT is a regulator
 - Decides what is the next thing the shell will execute



Just in time parallelization

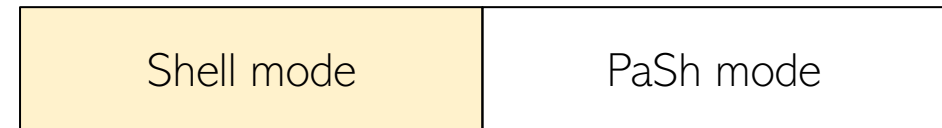
```
OUT=${OUT:-$TOP/out}
for input in $(ls ${IN}); do
  cat "$IN/$input" |
  tr -sc '[A-Z][a-z]' '[\012*]' |
  sort > "${OUT}/${input}.out"
done
```

Shell mode	PaSh mode
------------	-----------

TOP=/pash
IN=/pash/in

Just in time parallelization

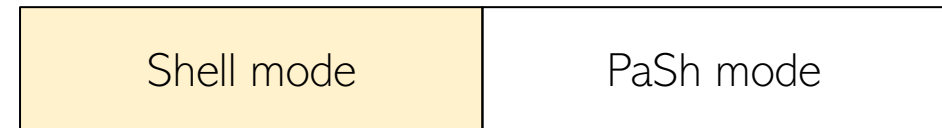
```
OUT=${OUT:-$TOP/out}
for input in $(ls ${IN}); do
  cat "$IN/$input" |
  tr -sc '[A-Z][a-z]' '[\012*]' |
  sort > "${OUT}/${input}.out"
done
```



TOP=/pash
IN=/pash/in

Just in time parallelization

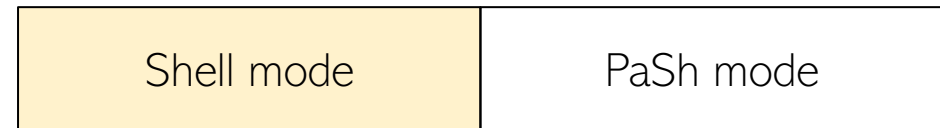
```
OUT=/pash/out
for input in $(ls ${IN}); do
  cat "$IN/$input" |
  tr -sc '[A-Z][a-z]' '[\012*]' |
  sort > "${OUT}/${input}.out"
done
```



```
TOP=/pash
IN=/pash/in
```

Just in time parallelization

```
OUT=/pash/out
for input in $(ls ${IN}); do
  cat "$IN/$input" |
  tr -sc '[A-Z][a-z]' '[\012*]' |
  sort > "${OUT}/${input}.out"
done
```



```
TOP=/pash
IN=/pash/in
OUT=/pash/out
```

Just in time parallelization

```
OUT=/pash/out
for input in in1 in2; do
  cat "$IN/$input" |
  tr -sc '[A-Z][a-z]' '[\012*]' |
  sort > "${OUT}/${input}.out"
done
```

Shell mode	PaSh mode
------------	-----------

```
TOP=/pash
IN=/pash/in
OUT=/pash/out
```

Just in time parallelization

```
OUT=/pash/out
for input in in1 in2; do
    cat "$IN/$input" |
    tr -sc '[A-Z][a-z]' '[\012*]' |
    sort > "${OUT}/${input}.out"
done
```

Shell mode	PaSh mode
------------	-----------

```
TOP=/pash
IN=/pash/in
OUT=/pash/out
input=in1
```


Just in time parallelization

```
OUT=/pash/out
for input in in1 in2; do
    cat "$IN/$input" |
    tr -sc '[A-Z][a-z]' '[\012*]' |
    sort > "${OUT}/${input}.out"
done
```

Shell mode	PaSh mode
------------	-----------

```
TOP=/pash
IN=/pash/in
OUT=/pash/out
input=in1
```

Expanding

Just in time parallelization

```
OUT=/pash/out
for input in in1 in2; do
  cat "$IN/$input" |
  tr -sc '[A-Z][a-z]' '[\012*]' |
  sort > "${OUT}/${input}.out"
done
```

Shell mode	PaSh mode
------------	-----------

```
TOP=/pash
IN=/pash/in
OUT=/pash/out
input=in1
```

Just in time parallelization

```
OUT=/pash/out
for input in in1 in2; do
  cat /pash/in/in1 |
  tr -sc '[A-Z][a-z]' '[\012*]' |
  sort > /pash/out/in1.out
done
```

Shell mode	PaSh mode
------------	-----------

```
TOP=/pash
IN=/pash/in
OUT=/pash/out
input=in1
```

Just in time parallelization

```
OUT=/pash/out
for input in in1 in2; do
    cat /pash/in/in1 |
    tr -sc '[A-Z][a-z]' '[\012*]' |
    sort > /pash/out/in1.out
done
```

Shell mode	PaSh mode
------------	-----------

```
TOP=/pash
IN=/pash/in
OUT=/pash/out
input=in1
```

Parallelize?

Just in time parallelization

```
OUT=/pash/out
for input in in1 in2; do
  mkfifo f1 f2 f3 f4
  cat /pash/in/in1 | split f1 f2 &
  ... &
  sort < f1 > f3 &
  sort < f3 > f4 &
  sort -m f3 f4 > /pash/out/in1.out
  rm f1 f2 f3 f4
done
```

Shell mode	PaSh mode
------------	-----------

TOP=/pash
IN=/pash/in
OUT=/pash/out
input=in1

Parallelize?
Success!

Just in time parallelization

```
OUT=/pash/out
for input in in1 in2; do
    mkfifo f1 f2 f3 f4
    cat /pash/in/in1 | split f1 f2 &
    ... &
    sort < f1 > f3 &
    sort < f3 > f4 &
    sort -m f3 f4 > /pash/out/in1.out
    rm f1 f2 f3 f4
done
```

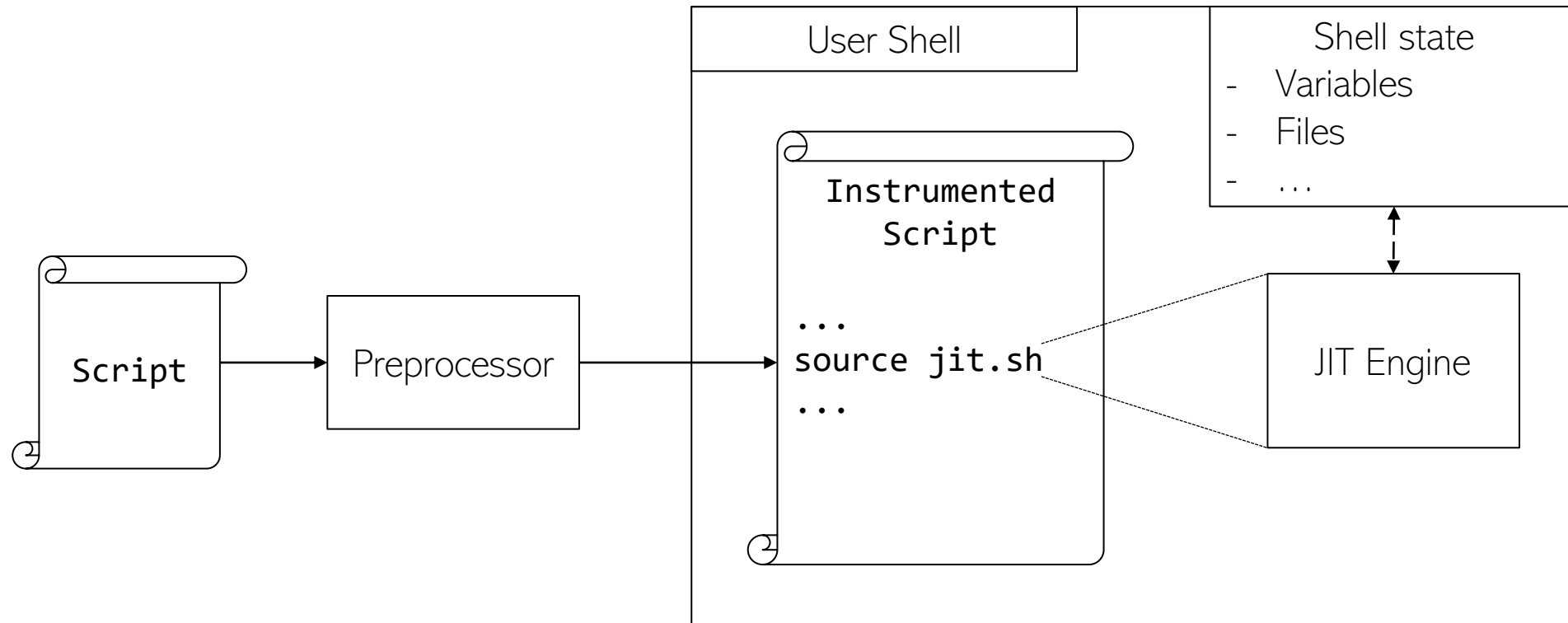
Shell mode	PaSh mode
------------	-----------

```
TOP=/pash
IN=/pash/in
OUT=/pash/out
input=in1
```

Challenge: How to JIT?

- Without modifying underlying interpreter
 - Cannot just add a branch in the bash interpreter loop
- Without being observable to user
 - Different shell configs like ``set -e`` expose info
- With minimal overhead
 - Frequent transitions between shell and pash mode

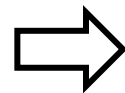
Solution Architecture



Preprocessor

- Replaces potentially parallelizable (DFG) regions with calls to the JIT
 - Intuitively, commands composed with pipes | and the background & operator
 - Stores original region for the JIT engine to access
- Preprocessing is optimistic; the actual decision is made at runtime
 - Syntactic analysis (no knowledge about commands and their behavior)

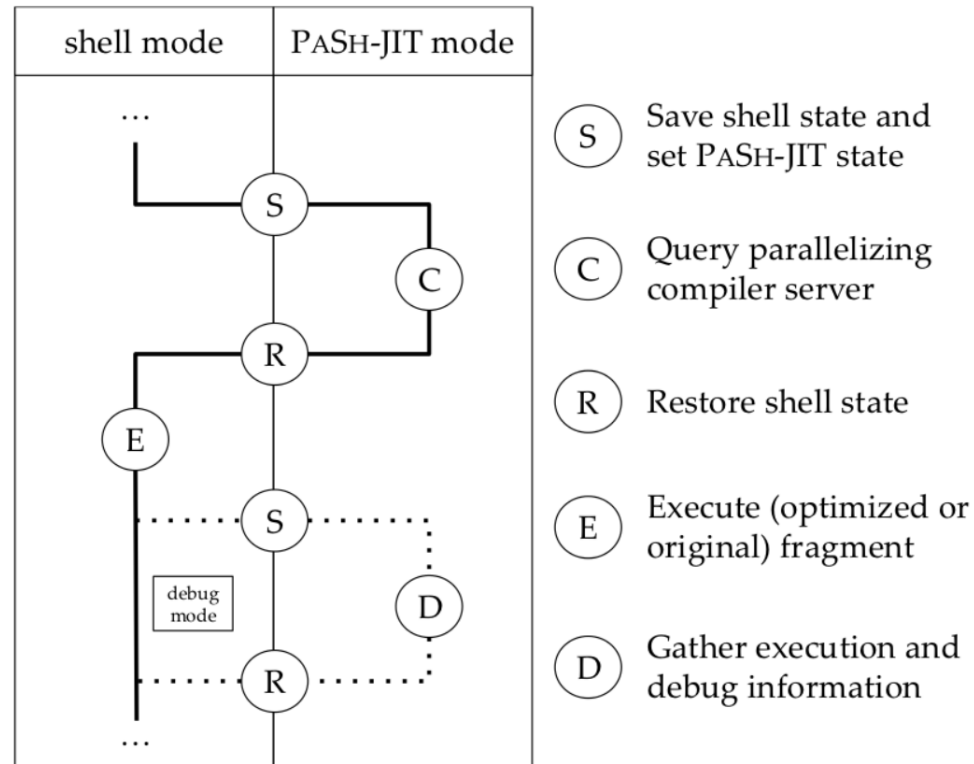
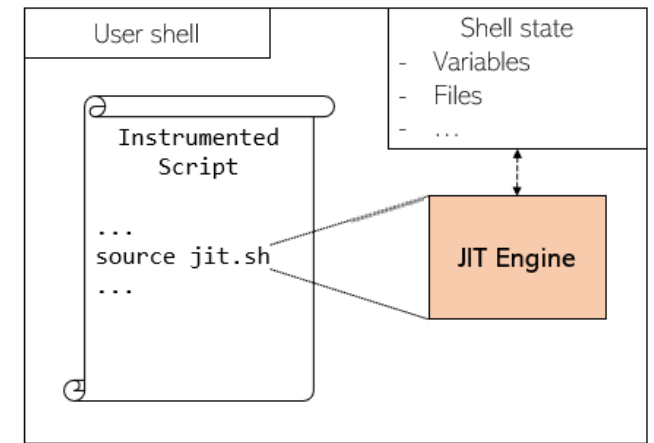
```
cd $TOP
OUT=${OUT:-$TOP/output}
mkdir -p "$OUT"
for input in $(ls ${IN}); do
    cat "$IN/$input" |
    tr -sc '[A-Z][a-z]' '[\012*]' |
    sort > "${OUT}/${input}.out"
done
```



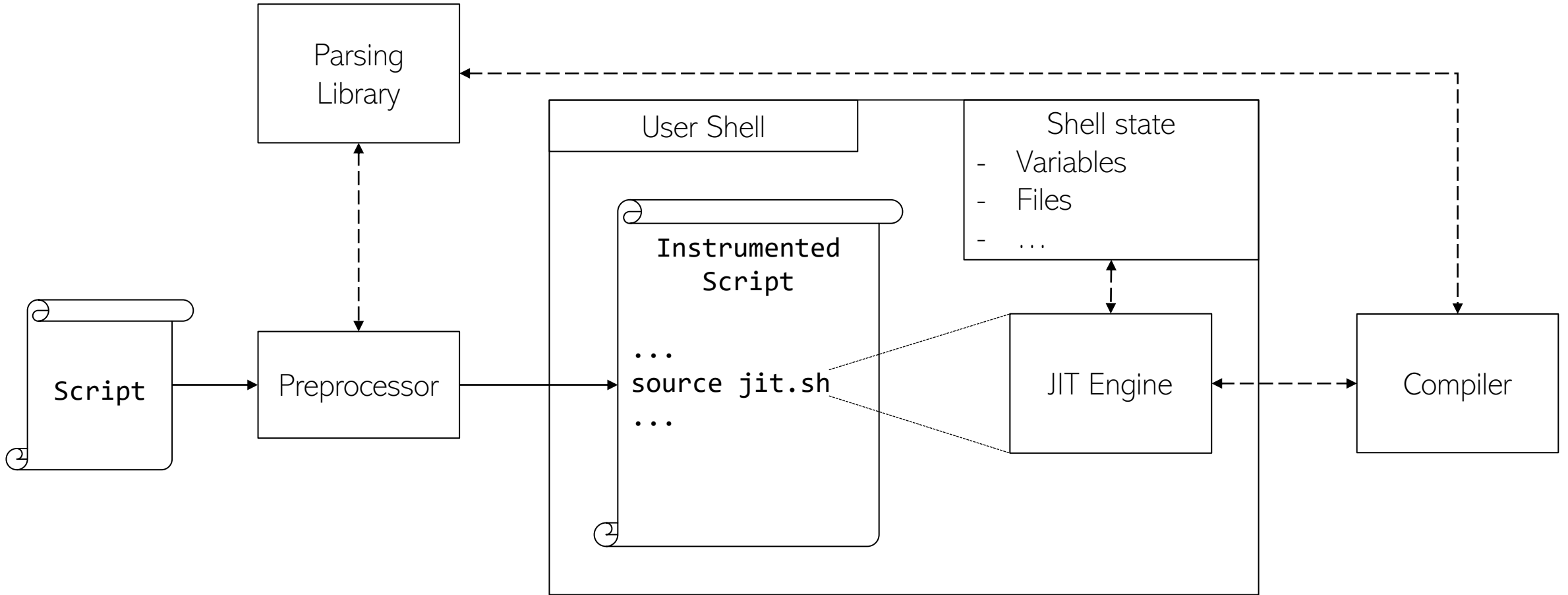
```
source jit.sh "$region8" # cd $TOP
OUT=${OUT:-$TOP/output}
source jit.sh "$region9" # mkdir -p "$OUT"
for input in $(ls ${IN}); do
    source jit.sh "$region10" # cat "$IN...
done
```

JIT Engine

- Hides compilation from the perspective of the shell
- Just a shell script
 - Heavily engineered for minimal latency

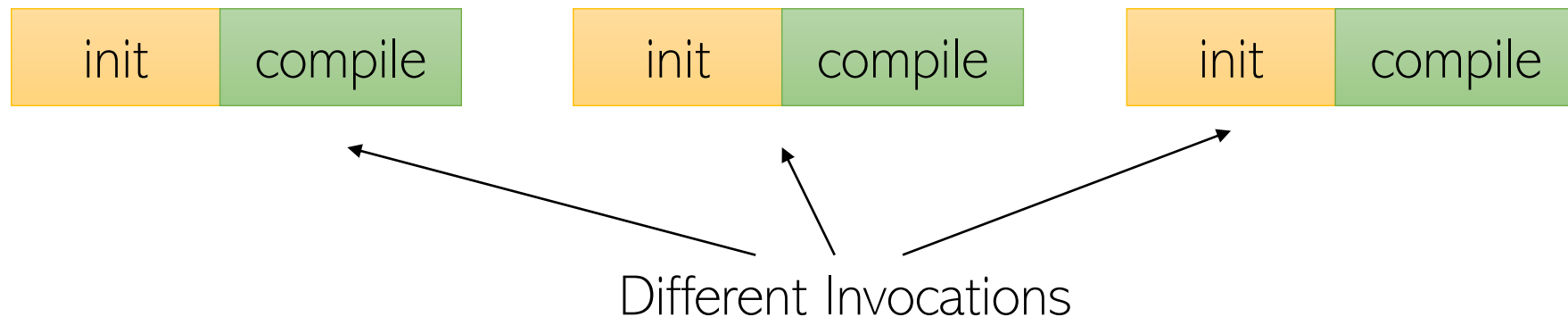


Complete PaSh-JIT overview



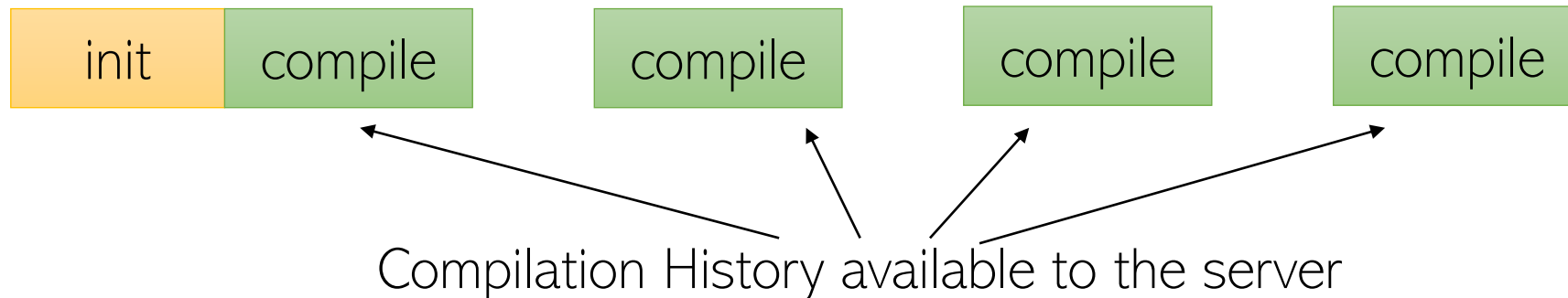
Compiler

- Original PaSh was a shell-to-shell compiler
 - Input script is compiled to a parallel output script
- Compiler is on critical path 🙄
 - Initialization happens on each invocation (10 – 1000 in a script)
- Work can not be reused across compiler invocations



Solution: Compilation Server

- Modify compiler to long-running compilation server
 - Communication through UNIX domain sockets
- Reduces latency!
 - Initialization happens once
- Also enables additional optimizations
 - Parallelization of independent fragments (e.g., iterations that touch different files)
 - Profile-guided optimizations (e.g., configuring parallelization width)

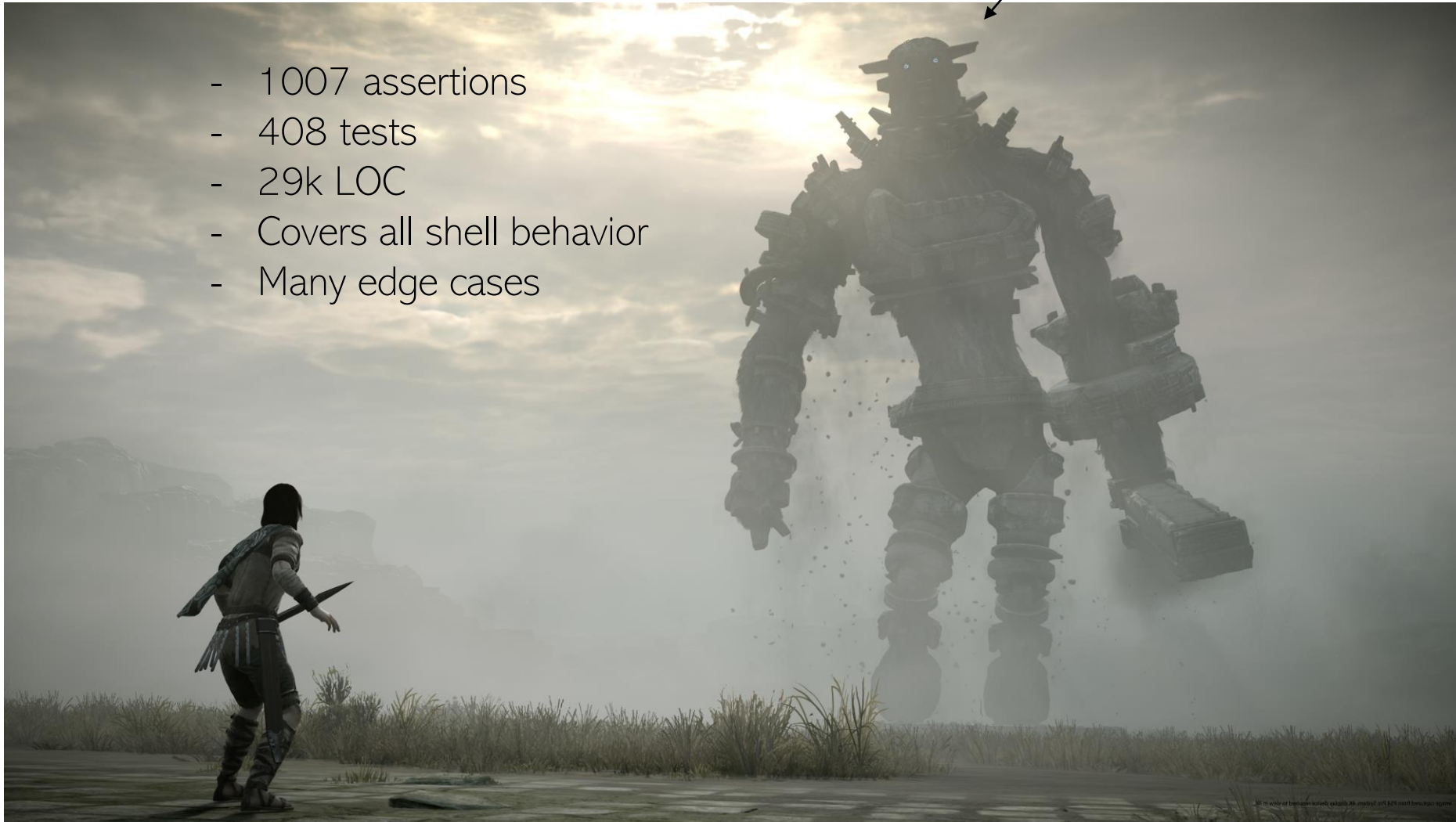


Evaluation

Evaluation: Correctness

POSIX Shell Test Suite

- 1007 assertions
- 408 tests
- 29k LOC
- Covers all shell behavior
- Many edge cases



Evaluation: POSIX test suite

- Out of the 408 tests
 - Bash passes 376 and fails 32 tests
 - PaSh-JIT passes 374 and fails 34 tests
- Divergence in these two tests is only in the exit status
 - Both return with an error, though different code
- Other shells compared to bash:
- Various shell failures on POSIX tests:

By following a lightweight shim approach (instead of reimplementing) we achieve very high compatibility with bash ✨



	Bash succeeds	X fails
dash	20	
ksh	22	
mksh	29	
posh	52	
yash	20	

Evaluation: Performance

- Evaluating on 82 shell scripts (4 suites and 11 standalone scripts)

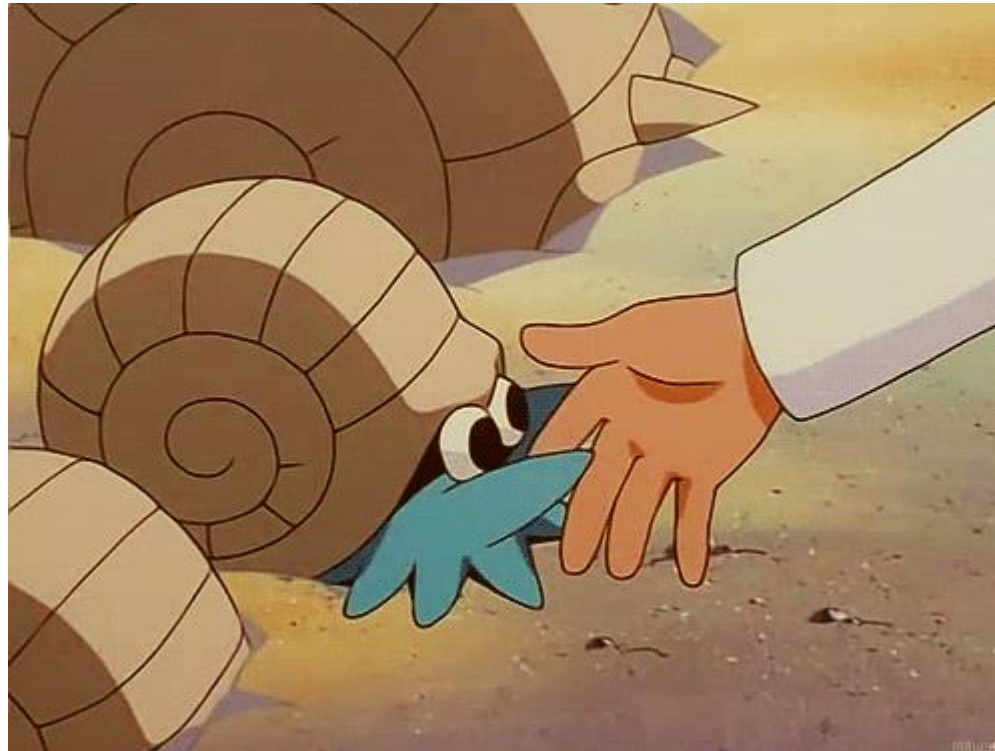


Avg speedups: PaSh-JIT (x5.8) – PaSh-AOT (x2.9)


Conclusion

Conclusion

- Shells were angry that we tried to parallelize statically
- We can make them happy by being dynamic
- Are we done?



The shell has more problems...

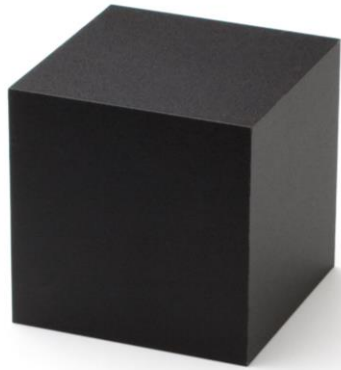
- Error-proneness
 - accidentally ``rm -rf /`` 
- Hard to learn
 - still googling for if-then-else shell syntax
- Redundant recomputation
 - we have to use Makefiles etc
- Lack of support for contemporary deployments
 - managing a distributed cluster

Recent exceptions: Rattle [1] and Riker [2]

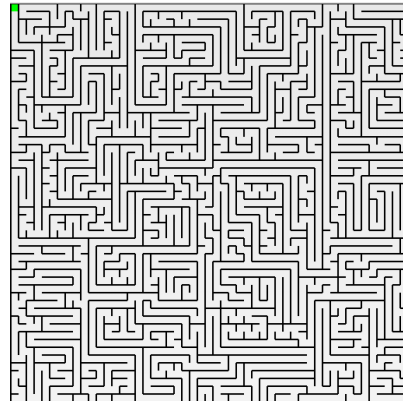
[1] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. “Build scripts with Perfect Dependencies.” OOPSLA. 2020.

[2] Charlie Curtsinger, and Daniel W. Barowy. “Riker: Always-Correct and Fast Incremental Builds from Simple Specifications” ATC. 2022.

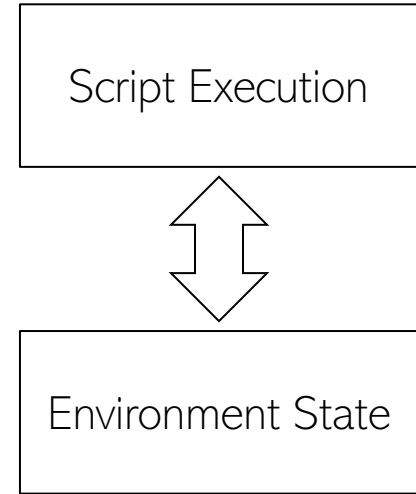
Challenging aspects of the shell



Lack of generality



Hard compliance

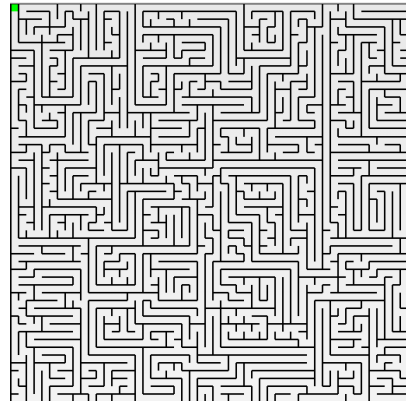


Lack of precision

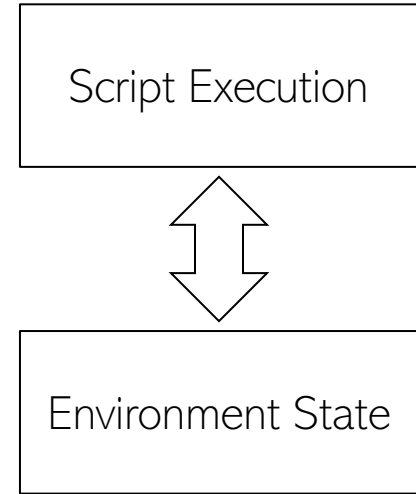
Our solutions



Command specifications

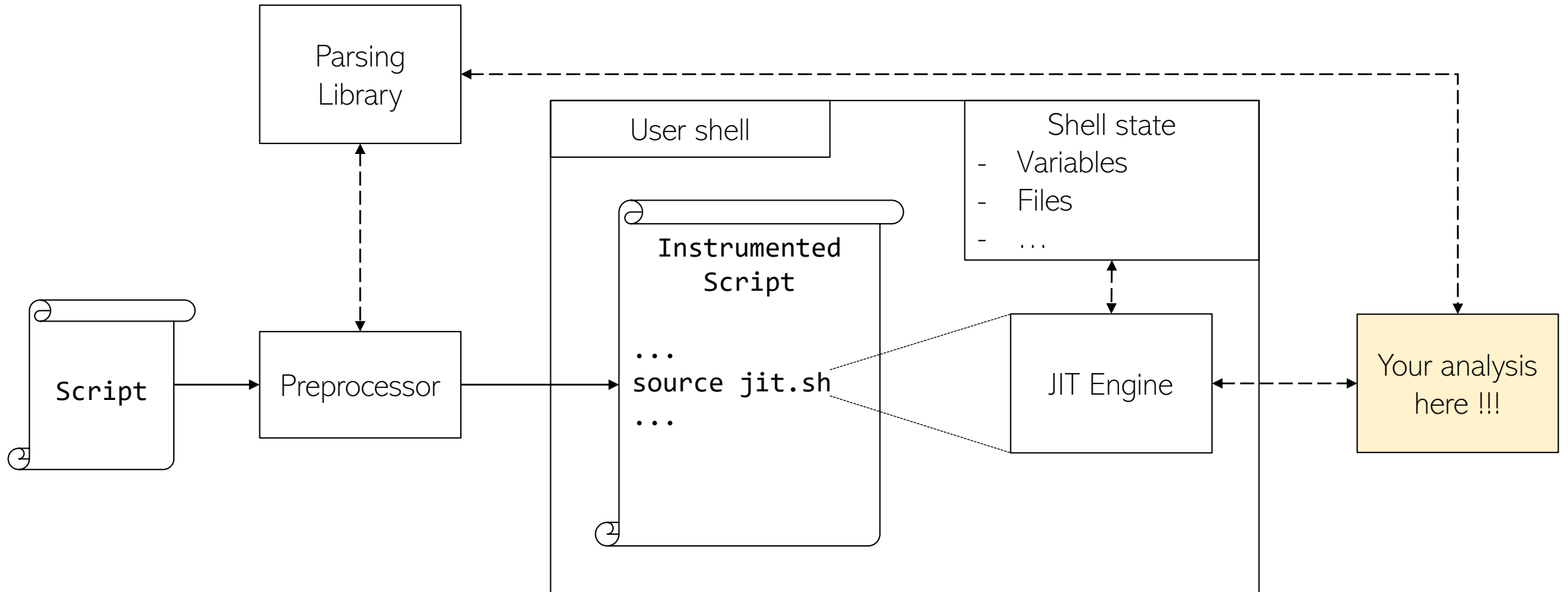


Shell to shell compiler



JIT architecture

Our infrastructure is an enabler

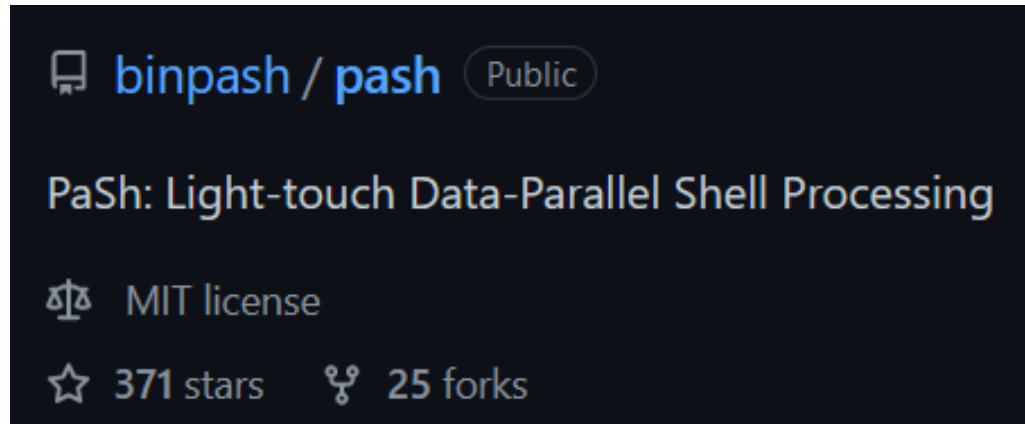



Some exciting future directions

- A shell monitor that ensures that safety/security props are not violated
- A fully distributed shell
- An incremental execution shell
- Talk to us if you have ideas!

Practical impact and availability

- PaSh is open source and hosted by the Linux Foundation



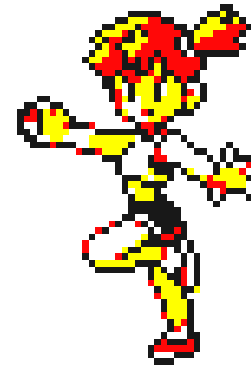
- It is virtually indistinguishable from bash (406/408 POSIX tests)
 - And requires no modifications/reimplementation
- Download it and play binpa.sh  github.com/binpash/pash

Backup slides

Wait, we have to go through them!!!



Arbitrary black-box
commands



Subtle Parallelism

How does PaSh do that?



Challenge: Arbitrary black-box commands



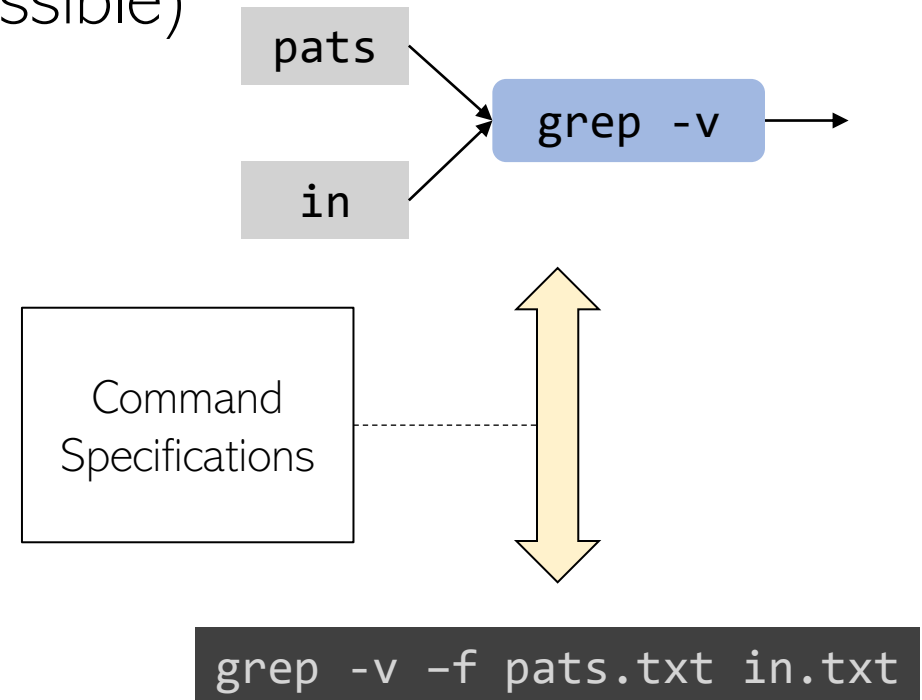
- Restricted programming frameworks (MapReduce, Spark, etc)
 - offer a limited set of constructs
 - can be easily mapped to a dataflow abstraction
- The shell is used to compose:
 - arbitrary commands
 - written in arbitrary languages
 - and are updated (or modified) over time
- This makes automated analysis infeasible
- Any one-time effort quickly obsolete and useless.

Solution: Command Specification Framework



For each command **cmd**, developers describe how to:

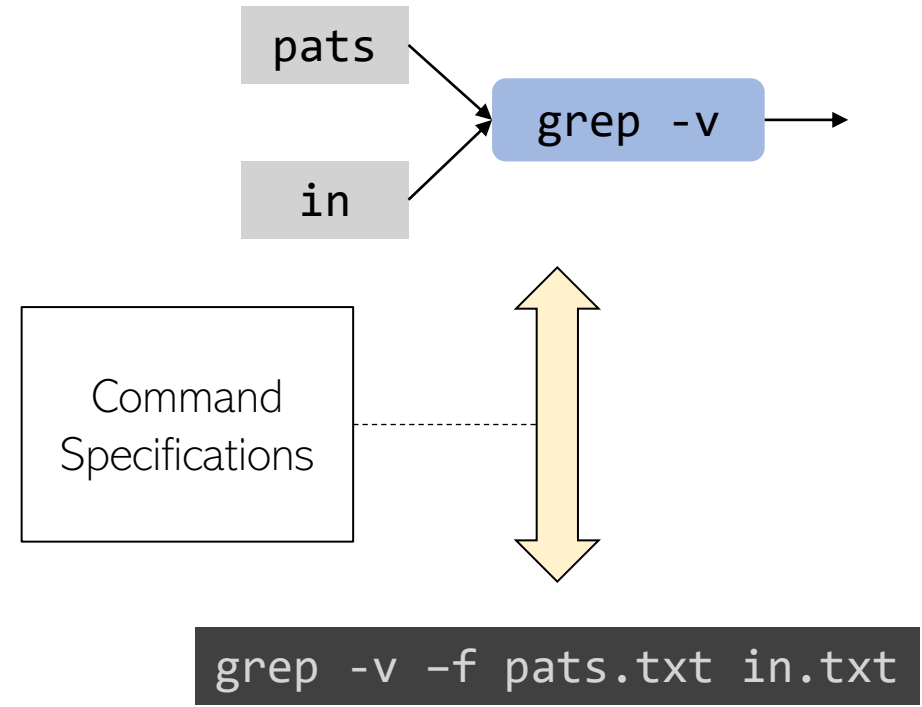
- Map its invocation to a dataflow node (if possible)
 - Inputs, outputs, parallelizability from arguments
- Map a dataflow node to a **cmd** invocation
 - Arguments from inputs, outputs, metadata
- By defining two python functions



Command Specs are shareable



- Developer instantiates correspondence **once** for each command
- The goal is for this to be used by
 - command developers or other experts
 - not users!
- Library of correspondence can be
 - inspected
 - shared





Command Spec Library

- We did a study of all POSIX and GNU Coreutils commands
- For POSIX, about 30% are pure
 - They write on a well-defined set of files and read from a set of files
 - The rest are side-effectful (e.g., mv, rm, chmod)
- Out of those, about 70% (~21% in total) are parallelizable
 - 2/3 are trivially parallelizable, we simply concatenate parallel output, e.g., grep
 - 1/3 need a non-trivial aggregator, e.g., sort needs a merge

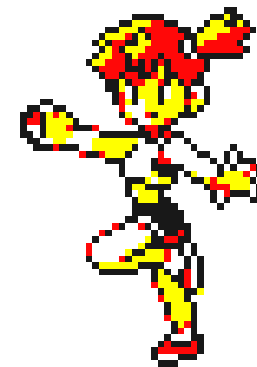
Class	Key	Examples	Coreutils	POSIX
Stateless	Ⓢ	tr, cat, grep	13 (12.5%)	19 (12.7%)
Parallelizable Pure	Ⓟ	sort, wc, head	17 (16.3%)	13 (8.7%)
Non-parallelizable Pure	Ⓝ	sha1sum	13 (12.5%)	11 (7.3%)
Side-effectful	ⓔ	env, cp, whoami	61 (58.6%)	105 (70.4%)

Solution: Node Correspondence Framework



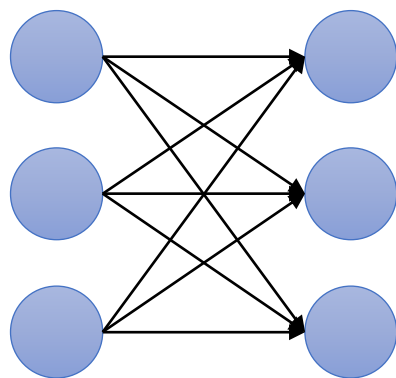
- Furthermore, most commands have restricted, well-defined behavior
- Designed an annotation language
 - Annotation uniquely defines the two correspondence functions
 - Language guided by the POSIX and GNU Coreutils study
- Part of annotation for cat:
- Defined annotations for 53 commands
- More details in our EuroSys 21 paper

```
{
  "command": "cat",
  "cases": [
    ...,
    {
      "predicate": "default",
      "class": "parallelizable",
      "aggregator": "cat",
      "inputs": ["args[:]"],
      "outputs": ["stdout"]
    }
  ]
}
```

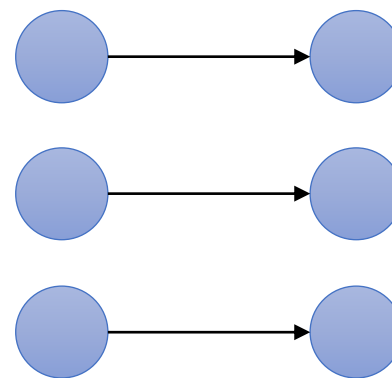


Challenge: Shell Data Parallelism is Subtle

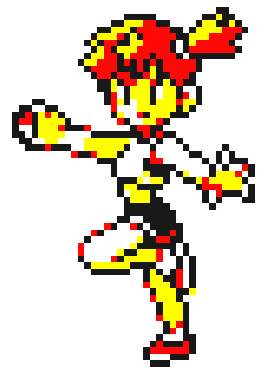
- Parallel frameworks such as MapReduce or Spark
 - Either require commutativity
 - Or key-by independence to achieve parallelism



Round Robin

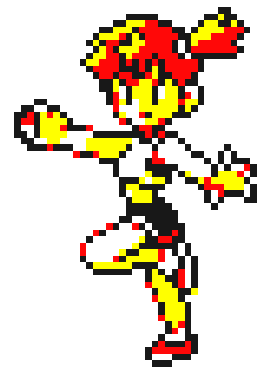


Partition By Key



Challenge: Shell Data Parallelism is Subtle

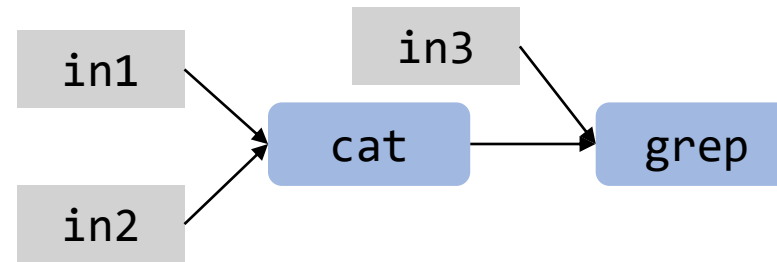
- Data parallelism in the shell is trickier
 - Commutativity and independence based on key is rare
- Commands most often read from their inputs in sequence
 - This order matters for the output
 - For example, `grep "foo" in1 - in2` reads in1, its stdin, and then in2
- We need a model that captures a parallelizable subset of the shell
 - That also captures order of command input consumption



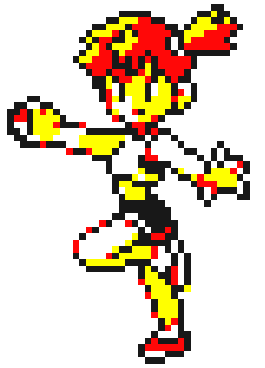
Solution: Order-aware dataflow model

- The following pipeline would be translated to:

```
cat in1 in2 | grep "foo" in3 -
```



- Model defines a shell fragment with no scheduling constraints
 - Roughly: commands composed with `&`, `|`
 - The dataflow ends when encountering `;`, `&&`, or control flow
- The expressiveness allows us to define a bidirectional correspondence between:
 - Shell programs in this fragment
 - Dataflow graphs in our model



Solution: Order-aware dataflow model

- On the graph we have defined semantics preserving transformations.

• M

$$\frac{\text{cmd2node}(w, \bar{x}_o \leftarrow f(\bar{x}_i)) \quad \text{add_metadata}(f, \bar{a}s, \bar{r}) = f' \quad \text{redir}(\bar{x}_o, \bar{x}_i, \bar{r}, \bar{x}'_o, \bar{x}'_i)}{\bar{a}s\bar{w}\bar{r} \uparrow \langle \text{input}\bar{x}'_i; \text{output}\bar{x}'_o; \bar{x}'_o \leftarrow f'(\bar{x}'_i), fg \rangle} \quad \text{COMMANDTRANS}$$

$$\frac{\text{cmd2node}(w, \perp)}{\bar{a}s\bar{w}\bar{r} \uparrow \bar{a}s\bar{w}\bar{r}} \quad \text{COMMANDID} \qquad \frac{c \uparrow \langle p, b \rangle}{c\& \uparrow \langle p, bg \rangle} \quad \text{BACKGROUNDDFG} \qquad \frac{c \uparrow c'}{c\& \uparrow c'\&} \quad \text{I}$$

$$\frac{c_1 \uparrow \langle p_1, bg \rangle \quad c_2 \uparrow \langle p_2, b \rangle}{c_1; c_2 \uparrow \langle \text{compose}(p_1, p_2), b \rangle} \quad \text{SEQBOTHBG} \qquad \frac{c_1 \uparrow c'_1 \quad c_2 \uparrow \langle p_2, bg \rangle \quad \text{opt}(p_2) \Downarrow c'_2}{c_1; c_2 \uparrow c'_1; (c'_2\&)} \quad \text{S}$$

$$\frac{c_1 \uparrow \langle p_1, fg \rangle \quad c_2 \uparrow \langle p_2, bg \rangle \quad \text{opt}(p_1) \Downarrow c'_1 \quad \text{opt}(p_2) \Downarrow c'_2}{c_1; c_2 \uparrow c'_1; (c'_2\&)} \quad \text{SEQBOTHFGBG}$$

$$\frac{c_1 \uparrow c'_1 \quad c_2 \uparrow \langle p_2, fg \rangle \quad \text{opt}(p_2) \Downarrow c'_2}{c_1; c_2 \uparrow c'_1; c'_2} \quad \text{SEQRIGHTFG} \qquad \frac{c_1 \uparrow c'_1 \quad c_2 \uparrow c'_2}{c_1; c_2 \uparrow c'_1; c'_2} \quad \text{SEQNONE}$$

$$\frac{c_1 \uparrow \langle p_1, b_1 \rangle, \dots, c_n \uparrow \langle p_n, b_n \rangle, \quad p'_1 \dots p'_{n-1} = \text{map}(\text{connectpipe}, p_1 \dots p_{n-1}) \quad p = \text{fold_left}(\text{compose}, p'_1 p'_2 \dots p'_{n-1} p_n)}{\text{pipe}|c_1 c_2 \dots c_n \& \uparrow \langle p, bg \rangle} \quad \text{PIPEBG} \qquad \frac{c_1 \uparrow \langle p_1, b_1 \rangle, \dots, c_n \uparrow \langle p_n, b_n \rangle, \quad p'_1 \dots p'_{n-1} = \text{map}(\text{connectpipe}, p_1 \dots p_{n-1}) \quad p = \text{fold_left}(\text{compose}, p'_1 p'_2 \dots p'_{n-1} p_n)}{\text{pipe}|c_1 c_2 \dots c_n \uparrow \langle p, fg \rangle} \quad \text{I}$$

Figure 5. A subset of the compilation rules.

• In

- Can be parallelized by applying the map on
- And then applying the aggregate

$$\frac{x_j \leftarrow \text{Unused}(I, O, \mathcal{E}), \mathcal{E}' = \mathcal{E}[x_j/x_i]}{I, O, \mathcal{E} \iff I, O, \mathcal{E}' \cup \{x_j \leftarrow \text{relay}(x_i)\}} \quad \text{RELAY}$$

$$\frac{x_s, x'_s \leftarrow \text{Unused}(I, O, \mathcal{E}), E = \{ \langle x_s, x'_s \rangle \leftarrow \text{split}(x), \langle x_1, \dots, x_k \rangle \leftarrow \text{split}(x_s), \langle x_{k+1}, \dots, x_m \rangle \leftarrow \text{split}(x'_s) \}}{I, O, \mathcal{E} \cup \{ \langle x_1, \dots, x_m \rangle \leftarrow \text{split}(x) \} \iff I, O, \mathcal{E} \cup E} \quad \text{SPLIT-SPLIT}$$

$$\frac{x_c, x'_c \leftarrow \text{Unused}(I, O, \mathcal{E}), E = \{ x_c \leftarrow \text{cat}(\langle x_1, \dots, x_k \rangle), x'_c \leftarrow \text{cat}(\langle x_{k+1}, \dots, x_m \rangle), x \leftarrow \text{cat}(\langle x_c, x'_c \rangle) \}}{I, O, \mathcal{E} \cup \{ x \leftarrow \text{cat}(\langle x_1, \dots, x_m \rangle) \} \iff I, O, \mathcal{E} \cup E} \quad \text{CONCAT-CONCAT}$$

$$\frac{\bar{x} \leftarrow \text{Unused}(I, O, \mathcal{E}), E = \{ \bar{x} \leftarrow \text{split}(x_j), x_i \leftarrow \text{cat}(\bar{x}) \}}{I, O, \mathcal{E} \cup \{ x_i \leftarrow \text{relay}(x_j) \} \iff I, O, \mathcal{E} \cup E} \quad \text{SPLIT-CONCAT}$$

$$\frac{x_1^u, x_1^d, x_2^u, x_2^d, \dots, x_n^u, x_n^d \leftarrow \text{Unused}(I, O, \mathcal{E}), \quad E = \{ \langle x_1^u, x_1^d \rangle \leftarrow \text{tee}(x_1), \langle x_2^u, x_2^d \rangle \leftarrow \text{tee}(x_2), \dots, \langle x_n^u, x_n^d \rangle \leftarrow \text{tee}(x_n), \quad x_o \leftarrow \text{cat}(x_1^u, x_2^u, \dots, x_n^u), x'_o \leftarrow \text{cat}(x_1^d, x_2^d, \dots, x_n^d) \}}{I, O, \mathcal{E} \cup \{ x \leftarrow \text{cat}(x_1, x_2, \dots, x_n), \langle x_o, x'_o \rangle \leftarrow \text{tee}(x) \} \iff I, O, \mathcal{E} \cup E} \quad \text{TEE-CONCAT}$$

$$\frac{x \leftarrow \text{Unused}(I, O, \mathcal{E}), \quad \bar{x}_i = \langle x_1, x_2, \dots, x_n \rangle, \bar{x}_j = \langle x'_1, x'_2, \dots, x'_n \rangle, E = \{ x'_1 \leftarrow \text{relay}(x_1), x'_2 \leftarrow \text{relay}(x_2), \dots, x'_n \leftarrow \text{relay}(x_n) \}}{I, O, \mathcal{E} \cup \{ x \leftarrow \text{cat}(\bar{x}_i), \bar{x}_j \leftarrow \text{split}(x) \} \iff I, O, \mathcal{E} \cup E} \quad \text{CONCAT-SPLIT}$$

$$\frac{I, O, \mathcal{E} \cup \{ x_j \leftarrow \text{cat}(x_i) \}}{I, O, \mathcal{E} \cup \{ x_j \leftarrow \text{relay}(x_i) \}} \quad \text{ONE-CONCAT}$$

$$\frac{I, O, \mathcal{E} \cup \{ x_j \leftarrow \text{split}(x_i) \}}{I, O, \mathcal{E} \cup \{ x_j \leftarrow \text{relay}(x_i) \}} \quad \text{ONE-SPLIT}$$

aggregate

- Auxiliary transformations enable parallelization by inserting cat + split.

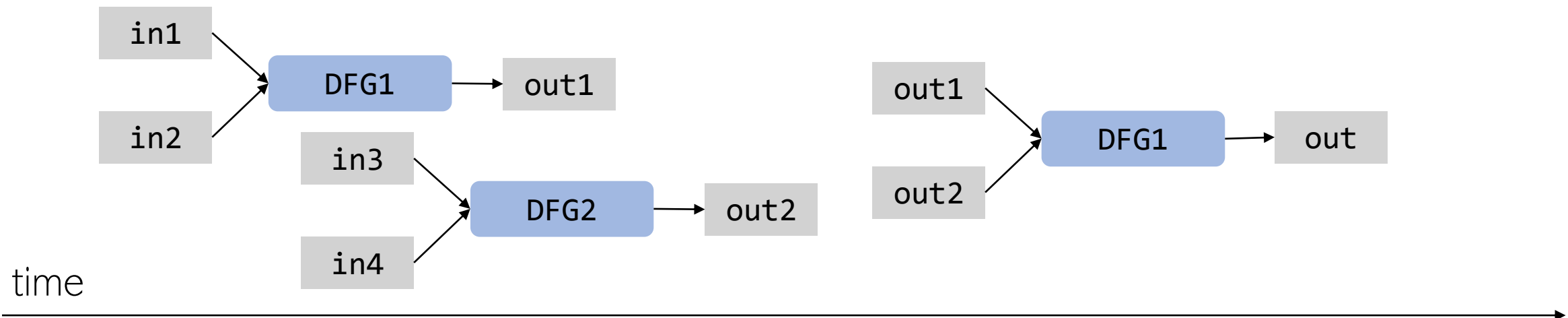
Proofs in our ICFP21 paper!

Dependency Untangling

- In the following script, each grep executes after the previous is done

```
grep "user1" in1 in2 > out1
grep "user2" in3 in4 > out2
grep "error" out1 out2 > out
```

- But the first two are completely independent
- Compilation server dynamically tracks dependencies
 - Allows independent regions to run in parallel



Teaser: Distribution

