RT: Regular Types for the Streaming Shell

Zekai Li
Brown University

Lukas Lazarek
Brown University

Evangelos Lamprou
Brown University

George Kapetanakis
Brown University

Konstantinos Mamouras
Rice University

Nikos Vasilakis
Brown University

Abstract

This paper presents an overlay type system, RT, for statically checking streaming shell programs or fragments *before* their execution. RT's *regular types* offer expressiveness appropriate for capturing a command's standard input and output streams, support computationally tractable and efficient type checking, and provide an interface encoded as regular expressions—*i.e.*, annotations and error messages familiar to developers versed in the Unix environment. RT's extensions around type polymorphism, finite-state transductions, environment concretization, and syntactic primitives offer additional expressiveness and improved precision. Applying RT to hundreds of programs from various sources including StackOverflow, GitHub, and prior literature indicates efficient type checking (0.02*s* on average), effectiveness at discovering serious bugs (90% accuracy), and key benefits from RT's extensions (up to 54% reduction in false negatives).

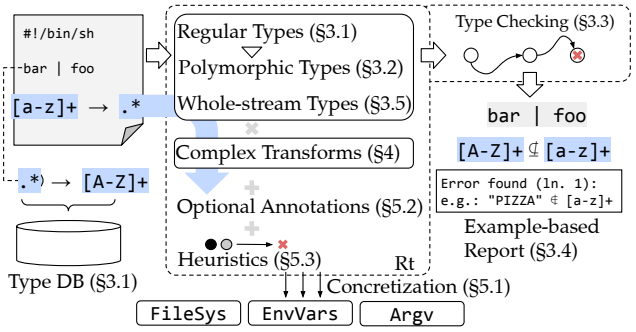


Fig. 1: RT overview. A shell program is parsed into RT, which assigns types to commands from a database of type declarations (§3.1). It models command behavior with simple regular types (§3.1), extends expressiveness through polymorphism (§3.2) and custom regular operations (§4), and detects composition errors with counterexamples (§3.5). Optional extensions—concretization (§5.1), annotations (§5.2), and heuristics (§5.3)—boost precision.

1 Introduction

The type systems of modern programming languages offer significant benefits [24, 32, 39, 46]: fast pre-execution checks for computations that may take days; detection of misbehaviors by a program's developer, not its user; improved error messages, including counter-examples; whole-program optimization opportunities, such as dead-code removal; and elimination of entire classes of bugs.

Unfortunately, these benefits are absent from the Unix shell—a pervasive environment used for a variety of tasks, ranging from system administration to data processing [20, 28, 31, 42, 47, 56]. Shell programs often combine commands developed in a variety of languages, interact in an untyped bytestream fashion, and are composed using the shell's intricate composition primitives. Coupled with the myriad of command flags and options, the assumptions about the structure of certain file contents (*e.g.*, `/etc/passwd`), and the ease by which stream contents are manipulated, these characteristics routinely lead to errors that with current practices can only be discovered *after* the execution of shell programs. Such errors, arising when programming-in-the-large and in-the-small, can have devastating effects—at best crashing the execution of a long-running task and at worst silently irreversibly corrupting entire filesystems [19, 43, 53, 54].

This paper proposes a new type system and associated implementation, RT, for statically checking shell programs before their execution. RT's novel *regular types* offer expressiveness appropriate for capturing the structure of a command's standard input and output streams, support computationally tractable and efficient type checking, and provide an interface encoded as regular expressions—*i.e.*, annotations and error messages familiar to developers versed in the Unix environment. With regular types, RT allows expressing, propagating, and checking constraints over the values of inter-process communication streams and the commands operating on them.

Regular languages are a constrained domain offering valuable computational and theoretical properties, but also meaning that not all command behaviors are directly representable;

```
1 find . |                        # Find files
2 grep 'book[0-9]+\.txt' |        # Filter by name
3 xargs cat |                     # Concat files
4 tr -cs A-Za-z '\n' |            # Split words
5 tr '[:lower:]' '[:upper:]' |    # Normalize
6 grep -fw dict.txt |             # Filter words
7 sort | uniq | sort -rn          # Freq sort
```

```
                ()  →  \.(/.+)?
                .*  →  .*book[0-9]+\.txt.*
[^␣\t]+|".+"|'.+'   →  .*
                .*  →  [A-Za-z]*
          [A-Za-z]* →  [A-Z]*
                    …
   ␣*[-+]?[0-9]+.*  →  ␣*[-+]?[0-9]+.*
```

Fig. 2: Left: A shell program with subtle composition mistakes. Right: Command types for each line in the left script; the last corresponds to `sort -rn`.

RT demonstrates that regular languages are sufficiently expressive to capture many interesting properties of command behavior. It effectively reasons about many commands via useful approximations and regular subsets of their domains, including handling commands manipulating data such as HTML or JSON. Furthermore, RT's regular-language reasoning about command input and output behavior allows it to identify bugs in computations that directly lead to dangerous side effects, such as pipelines computing path names to delete. By identifying bugs in such input-output manipulations, RT can prevent a wide array of bugs relating to dangerous side-effects without reasoning about those effects directly.

Figure 1 presents an overview of RT. RT's type system (§3) and checking algorithm (§3.4) come with several extensions such as polymorphic (§3.2) and whole-stream (§3.6) types aimed at improving precision while maintaining tractability. Additional extensions such as regular language operators and finite state transducers further improve expressiveness and accuracy (§4). The RT implementation is parameterized over a database of type declarations for every supported command invocation, akin to a typed standard-library in other languages. RT can optionally be configured to leverage developer annotations, additional heuristics, and environment concretization to improve its results (§5).

We evaluate RT on hundreds of shell programs across 8 suites, including dozens of real-world bugs collected from GitHub and StackOverflow. RT achieves 90% accuracy (855 out of 954) at classifying programs as correct or buggy. It uncovers 84 bugs ignored by state-of-the-art systems, including newly discovered bugs on production shell programs previously considered correct. RT's extensions offer valuable benefits in several cases—for example, finite state transducers quarter its false positive rate (from 20% to 5%). And it is fast: RT typechecks 954 programs in 20s, averaging 0.02s per program.

The paper starts by exemplifying the kinds of composition problems often found at the streaming core of the shell and how RT exposes them ahead of time (§2). Sections §3–§5 highlight RT's key contributions:

- *Regular types*: A type system targeting stream contents and commands operating on them, along with an efficient type-checking algorithm that automatically detects composition errors in shell programs (§3).
- *Regular language operators*: An extended set of regular language operations for common input-output transformations implemented efficiently via automata constructions (§4).
- *Extensions and optimizations*: Extensions and optimizations improving RT's precision—including a concretization subsystem for probing the current runtime context, lightweight annotation language for expressing developer intent, and heuristics for subtle composition errors (§5).

The paper continues with RT's evaluation (§6), discussion of related work (§7), and conclusion (§8). Appendices provide additional details on RT's FST construction technique (§B), a few extended examples showing how RT analyzes larger programs (§C), and definitions of RT's optional syntax sugar (§A).

2 Motivating Example

Consider the shell script in Fig. 2 a modern equivalent to McIlroy's classic one-liner [6]. It counts frequencies of correctly spelled words across all book files anywhere within the file tree rooted at the user's current directory.

Unfortunately, it does not work right: it produces error output saying that some files do not exist, and nothing on standard-out. While ShellCheck—a state-of-the-practice linter [18]—reports no warnings, RT uncovers several issues causing the undesired behavior. Its output begins:

```
$ rt spell.sh
  Error (ln. 2):
  > grep 'book[0-9]+\.txt' | xargs cat ...
        grep > (\.(/.+)?)&(.*book[0-9]+\.txt.*)
  maybe incompatible w/
   xargs cat > [^ \t]+|".+"|'.+'
  Counterexample: "./ book0.txt" ...
```

Regular types: RT reasons about composition mistakes by specifying the contents of communication channels with regular types. Regular *stream* types describe the language of the *lines* in the stream—*e.g.*, `find .` outputs paths relative to the current directory, one per line, so its output type is `\.(/.+)?`. Regular *command* types describe the languages of input-output streams—*e.g.*, `find .` :: () → `\.(/.+)?`.

To reason about Fig. 2's pipeline, RT leverages the corresponding command types listed in Fig. 2.[1]. By checking that the output language of every stage is included in the expected input of the following, RT reasons that the output language of `grep` (ln. 2) is not included in the expected input of `xargs cat` (ln. 3):

$$\texttt{.*book[0-9]+\.txt.*} \notin \texttt{[^␣\t]+|".+"|'.+'}$$

This is because filenames with spaces (*e.g.*, `my book1.txt`) are problematic inputs for `xargs cat`, as they are split into multiple arguments leading them to be misinterpreted as separate files (as in `cat my book1.txt`). The fix: `grep`'s pattern should be stricter to eliminate unexpected paths (`grep '^\./book[0-9]+\.txt$'`).

Configurability: Unfortunately, RT cannot reason precisely about the output stream type of `grep -f dict.txt` without the contents of `dict.txt`. Without information about the contents of the file, the best RT can say is that the patterns could be anything, so the invocation's output could be any subset of the language of its inputs (`[A-Z]*`)—not enough information to determine a problem.

However, RT can perform *environment-based concretization* by reading the contents `dict.txt` and computing a highly precise type for the command capturing every word in the file. The intersection of the input language of the `grep -f dict.txt` invocation (`[A-Z]*`) with the disjunction of every word in the file simplifies to the empty language (*i.e.*, the patterns can never match any input lines), thereby identifying another error.

With that information from RT, the problem and corresponding fix are apparent. The arguments of line 5's `tr` are swapped: it should normalize words to lowercase, not upper. The fixed line 5 is: `tr '[:upper:]' '[:lower:]'`.

Extensions: Fig. 2 omits simple regular types for `sort` and `uniq` by design. They pose a problem: what are their appropriate types? Consider `sort`, which only reorders lines, and therefore by the definition of stream types, accepts any type and produces output of the same type. Hence, the most precise input type for `sort` would be whatever `grep -f dict.txt` produces, and its output would be the same.

The trouble with `sort` reveals a general concern, shared by `grep`, `uniq`, and others: the most precise type for a particular use of the command depends on its context. To enable succinctly expressing types with this property, RT supports two extensions to regular types: *polymorphic command types* and *domain-specific type transformation operations*. With these extensions, RT's type database can provide equivalent or more precise versions of all of the types in Fig. 2 as well as (for the final program): `[a-z]* → [a-z]*` for `sort`, and `[a-z]* → ␣*[0-9]+ [a-z]*` for `uniq -c`.

[1]These are not the only types that could accurately summarize each invocation's behavior—in general, there may be infinitely many possible types for any given invocation, all of which may be correct (if not precise), see §3.

$$\begin{array}{rcl} T & ::= & \textit{literal} \mid \textit{character class} \mid \texttt{.} \mid \texttt{\^{}} \mid \texttt{\$} \\ & \mid & TT \mid T\texttt{*} \mid T\texttt{+} \mid T\texttt{?} \mid \texttt{(}T\texttt{)} \mid T\texttt{|}T \\ & \mid & T\{n\} \mid T\{n,\} \mid T\{n,m\} \\ & \mid & \texttt{!}T \mid T\texttt{\&}T \\ n & \in & \mathbb{N} \end{array}$$

Fig. 3: Base regular type grammar accepted by RT. The grammar is a superset of POSIX ERE, with the addition of negation (`!`) and intersection (`&`). The languages it describes are all regular.

Results: RT is able to identify several composition mistakes in about 0.02s (§6). Once corrected, the program triggers no additional RT warnings.

3 Regular Language Types

RT summarizes stream contents using regular stream types, and command input/output behavior using regular command types. This section details the nature of these types, and how RT uses them to automatically identify composition mistakes.

A regular stream type is a regular language, written as a regular expression, that describes the set of all possible contents of each line in a stream, mirroring the line-oriented nature of most UNIX commands. For example, a stream type describing the output of `find .` is `\./.*)?`, capturing that every line of output is a dot followed by an arbitrary string.

At its core, commands in RT are described with two stream types: an input and an output stream type. Intuitively, a regular command type describes a transformation of an input language to its image under the command. For example, the regular command type `.* → [0-9]+` describes how `wc -l` transforms the input language (anything) to the output language (decimal digits).

3.1 Regular Types and Commands

RT summarizes byte-stream contents using stream types—regular languages defined via extended regular expressions (EREs) [55] (see Fig. 3). These include standard ERE features such as concatenation, alternation (`|`), repetition (`*`, `+`, `?`), grouping with parentheses, character classes (e.g., `[A-Z]`, `[:digit:]`), anchors (`^`, `$`), and bounded quantifiers (e.g., `a{2,5}`).

RT does not support non-regular features like backreferences (found in PCRE [3]) so that language inclusion remains decidable [15]. To improve expressiveness while retaining regularity, RT adds two operators: intersection (`&`) and negation (`!`). For instance, `[a-z]+ & err.*` matches all lowercase words with a "err" prefix, and `!(abc)` matches any string not matching `abc`. All supported operations are closed under regular languages.

RT defines a stream type as *sound* if it accepts all possible stream contents, and *precise* if it accepts only those that can

actually occur.

Command types: RT summarizes a command's input/output behavior using command types, which pair stream types for each stream. A command type is sound if its input type is a subset of the command's safe domain—the set of inputs guaranteed not to cause crashes—and its output type is a superset of the command's possible outputs.

Although many UNIX commands don't correspond directly to regular-language transformations, most can be approximated with sufficient accuracy. For instance, `grep`'s output is effectively the intersection of its input with a search-language. In contrast, commands like `sort` exceed the expressiveness of regular types, but can still be usefully approximated—*e.g.*, by assigning identical input/output types that abstract over line content, since the set of lines remains unchanged.

Commands like `tee`, `paste`, or `comm` involve multiple streams; RT assigns each a separate type. For simplicity, this paper discusses commands in terms of a single input and single output stream unless otherwise noted.

The type database: RT is parameterized over a database of command type declarations for supported invocations, much like how the type system of a typical programming language requires a manually-crafted base type environment declaring the types of primitive runtime functions. In general, RT's type database could be a literal mapping from invocations to command types, or arbitrary functions that produce a command type given an invocation. The RT prototype strikes a middle-ground, with manually-written configuration files that define (1) how to parse relevant invocations and (2) how to construct a command type given an invocation using parsed invocation flags and arguments (useful for example for `grep`, whose output type can be constructed from its argument regular expression). For commands not found in the database, RT defaults to the command type that is trivially correct for all command invocations: `.* → .*`. In total, RT's starting database covers 71/106 GNU core utils (see §6 for more details).

3.2 Polymorphic Regular Types

RT overcomes the challenge of precisely and succinctly describing the (infinitely) many types of commands like `cat` by introducing *polymorphic* command types. In particular, command types may be parameterized over *type variables*, which stand in for any possible regular language, and which can be used as the input of a command type, and anywhere within the output type. The polymorphic type of `cat`, for example, can be written $\forall\alpha.\alpha \to \alpha$—capturing the fact that the output language of `cat` is exactly the same as its input.

While some commands, like `cat`, impose no expectations on their input, many commands with polymorphic types do: they are polymorphic over a constrained set of languages (analogous to bounded polymorphism [9]). For instance, `sort -n` copies its input to its output (abstracting across all lines), but its numeric sorting expects each input line to begin with decimal digits. Polymorphic command types can express constraints over polymorphic variables with a notation familiar from mathematics, such as the following type for `sort -n`: $\forall\alpha \subseteq$ `[␣\t]*[-+]?[0-9]+.* .` $\alpha \to \alpha$.

Polymorphic command types do not introduce any challenges for type-checking. RT reasons about polymorphic command types after concretizing them into simple types, so that type compatibility checks are always over simple stream types.

3.3 Dataflow Analysis

Shell scripts go beyond simple linear pipelines: they assign intermediate results to variables via command substitution, expand those variables as arguments in later commands, and compose fragments across subshells and sourced scripts. RT performs dataflow analysis to track how values flow through these constructs, constructing a dataflow graph that enables type-checking entire scripts.

Given a shell script, RT constructs a dataflow graph—a DAG whose nodes represent computation stages and whose edges represent data dependencies between them. Beyond ordinary command invocations, RT introduces additional nodes to model the shell constructs that define and transform variable values.

Command substitution nodes type-check the enclosed pipeline and assign its output type to the target variable, which then flows to all subsequent uses of that variable. For example, in:

```
IP=$(ip -4 addr | sed ... | awk '{print $1}' |
↪  head -1)
iptables -t nat -A PREROUTING -p udp -d $IP ...
```

RT infers the output type of the pipeline, assigns it to `$IP`, and checks it against the type expected by `iptables -d` (a valid network address) automatically.

Shell expansion nodes model unquoted variable references, which include word splitting and globbing; RT overapproximates the expansion soundly. Expansion nodes also impose implicit constraints: in `rm -rf $prefix/bin`, the unquoted `$prefix` undergoes word splitting, so RT constrains `$prefix` to not contain whitespace—a constraint arising from the expansion semantics rather than any command's type declaration. Arithmetic expansion nodes constrain their operands to be numeric and produce a numeric result type. Simple variable assignments set the variable's type directly; when a variable is assigned in multiple branches (*e.g.*, both branches of an `if`), RT takes the union of the branch types. Subshells are inlined into the graph, preserving type information, and sourced scripts are similarly inlined when the script path can be statically resolved.

The resulting dataflow graph is a DAG, which RT type-checks using the algorithm described next.

Algorithm 1: RT type-checking algorithm. Given a dataflow graph (§3.3), RT iterates over nodes in topological order, checking if each node's input is compatible with its type declaration and propagating output types to downstream nodes. For a pipeline $c_1|c_2|\ldots|c_n$, the graph is a linear chain.

```
1  Function CheckDAG(dag, in):
2      dag.entry.input ← in;
3      for node ∈ TopoSort(dag.nodes) do
4          type ← GetTypeAnnotation(node);
5          if not CheckInput(type, node.input) then
6              ReportMismatch(type, node.input);
7          output ← InferOutputType(type,
             node.input);
8          for child ∈ node.children do
9              child.input ← output;
10 Function CheckInput(type, input):
11     if type is simple then
12         return IsSubset(input, type.input);
13     else
           // Polymorphic type annotation
14         return IsSubset(input, type.constraint);
15 Function InferOutputType(type, input):
16     if type is simple then
17         return type.output;
18     else
19         return type.output[type.var ↦ input];
```

3.4 Type Checking with Regular Types

Alg. 1 outlines the core of RT's type-checking algorithm. At a high level, RT iterates over the nodes of a dataflow graph (§3.3) in topological order, checking if each node's input is compatible with its type declaration and propagating output types to downstream nodes.

In detail, the main `CheckDAG` procedure accepts a dataflow graph *dag* and an initial input type *in*. It traverses the graph in topological order; for each node, it retrieves the command type with `GetTypeAnnotation`, checks if the node's accumulated input is compatible using `CheckInput`, then computes the output type with `InferOutputType` and propagates it to all of the node's children. For a simple pipeline $c_1|c_2|\ldots|c_n$, the dataflow graph is a linear chain and `CheckDAG` reduces to iterating over stages in sequence. The initial input type *in* is configurable: it can be the empty language (assuming no input) or `.*` (assuming arbitrary input), alongside other aspects of RT ((*Cf.*§5)).

The `CheckInput` and `InferOutputType` helpers check type compatibility and compute output types respectively. `CheckInput` consists of two cases: 1) if the command type is "simple" (*i.e.*, not polymorphic), then it checks if the input is a subset of the command type's input stream type—*i.e.*, regular language inclusion; 2) if the command type is polymorphic, then it checks if the input is a subset of the constraint bound on the command type's polymorphic input type. Similarly, `InferOutputType` mirrors the same structure with two cases: 1) if the command type is simple, then it returns the literal output stream type of the command; 2) if the command type is polymorphic, then it *concretizes* the output of the command type to obtain a simple stream type. Concretizing the polymorphic type means syntactically substituting the type variable with the actual input type. For example, for a polymorphic command type $\forall\alpha.\alpha \rightarrow$ `hi␣`α, and input type `[a-z]+`, the substitution in the output type results in the concrete type `hi␣[a-z]+`.

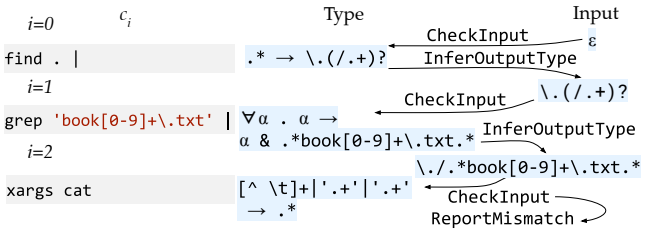


Fig. 4: Type-checking example. Applying alg. 1 to Fig. 2's pipeline, step by step. From left to right, the first column shows the command in the pipeline, the second column shows the type produced by `GetTypeAnnotation`, and the third column shows the intermediate stream type propagated to the next node after each iteration. From top to bottom the rows show the first, second, and third iterations of the loop in `CheckDAG`.

3.5 Error message generation

If the type checking algorithm detects type incompatibility between the input and the command in the pipeline, it generates error messages to help users understand and fix the composition error. The error message includes a witness x that demonstrates the type incompatibility, where x belongs to the input type but not to the expected type (for simple annotations) or the constraint type (for polymorphic annotations). Based on the set-theoretic equivalence: $x \in L_1 \wedge x \notin L_2 \leftrightarrow x \in L_1 \bigcap \overline{L_2}$, RT finds a witness from the automata of $L_1 \bigcap \overline{L_2}$.

3.6 Whole-stream Reasoning

While RT's stream types describe the language of every line in a stream, an alternative perspective is to describe the language of the entire stream's contents. This can be especially useful for commands that expect input or produce output in a format with a specific sequence of lines. For instance `ps` produces output beginning with a single header line, and all remaining lines follow a different format.

RT supports whole-stream types and converting between line-based and whole-stream representations. In particular, RT supports types describing multiple lines, and defines a stream

$$
\begin{array}{rcl}
T & ::= & \ldots \\
 & | & \texttt{reverse}(T) \\
 & | & \texttt{translate-match}(T, T, s, \texttt{global=}b) \\
 & | & \texttt{line-extract}(T, T) \\
 & | & \texttt{translate-chars}(T, C, C, \texttt{invert=}b, \texttt{squeeze=}b) \\
 & | & \texttt{field-select}(T, C, N, \texttt{invert=}b) \\
s & ::= & \textit{string} \mid \backslash n \mid s\,s \\
b & ::= & \texttt{true} \mid \texttt{false} \\
C & ::= & c \mid c-c \mid C,C \mid \textit{character class} \\
c & ::= & \textit{literal} \\
N & ::= & n \mid n-n \mid n- \mid N,N \\
n & \in & \mathbb{N}
\end{array}
$$

Fig. 5: Extended RT stream type grammar. The full syntax of regular stream types, extending regular expressions with five new primitive operations. This grammar extends the base regular type grammar RT accepts (Fig. 3)

as satisfying the type, if the sequence of complete lines in the stream until end of stream are included in the language (where the end of stream at the end of a non-empty line is equivalent to a newline followed by end of stream). Under this definition, any line-based type `T` can be converted to a whole-stream type as `(T\n)*`; RT can also convert any whole-stream type into a line-based one by unioning the language of all possible lines (*i.e.*, all sublanguages bounded by newlines, beginning, or end of stream). Hence, the latter conversion is a significant but unavoidable approximation. Theoretically, this approach is complicated by the possibility of infinite streams, but it suffices for the finite streams arising in the vast majority of practical shell scripts.

4 Complex Transformations

While simple and polymorphic types suffice to precisely model many commands—and approximate others—some approximated commands perform fine-grained manipulations that are (1) critical for reasoning about program correctness and (2) amenable to precise modeling. To capture such behaviors, RT leverages finite-state transducers to express complex regular-language transformations corresponding to common command semantics.

RT exposes these transformations as new primitive operations that extend the language of regular types beyond standard operations like union, intersection, and negation. This enables precise modeling of commands like `tr`, `cut`, and `grep -o`, whose behaviors are regular but not expressible using standard regular operations alone.

For instance, `grep '[A-Z]'` can be precisely typed using intersection and polymorphism, but `tr A-Z a-z` cannot—its transformation, while regular, lies outside the expressive power of standard operations. Furthermore, naive approximations using only standard operations can be unsound: modeling `tr -s ' '` as $\forall\alpha.\alpha \to \alpha$ `&!(.*␣␣.*)` underapproximates its output. With input `(␣b␣)`, this results in `(␣b␣)?`, whereas the actual output is `(␣(b␣)+)?`.

4.1 New Regular Language Operations

To improve precision while maintaining soundness, RT supports concisely expressing regular language types for these kinds of transformations via a novel set of regular language operations that, alongside standard operations, make up a domain-specific type description language. Fig. 5 shows the now-extended language of regular expressions.

The semantics of each operation is that of a type-level transformation of regular languages. The `reverse(`T_i`)` operator transforms language T_i into the reverse language, as recognized by the reversed DFA. `translate-match(`T_i`, `T_p`, `s`, global=`b`)` transforms input language T_i into a corresponding language where all leftmost-longest occurrences of T_p are replaced with s (if b is true, otherwise only the first)—and s may include group references (not to be confused with backreferences in patterns). The `line-extract(`T_i`, `T_p`)` operator transforms T_i into the sublanguage which also matches T_p. The `translate-chars(`T_i`, `C_f`, `C_t`, invert=`b_v`, squeeze=`b_s`)` operator transforms T_i into a corresponding language where all characters in C_f are replaced with the same-position character in C_t (where positions beyond the end of C_t translate to the last character of C_t, as in `tr`); if b_v is true, the set of characters to be replaced is the complement of C_f; if b_s is true, the result language also compresses repeat occurrences of each individual character in C_t to a single occurrence. The `field-select(`T_i`, `c`, `N`, invert=`b`)` operator transforms T_i into the language of just fields at indices N (or all other fields, if b is true), separated by c; fields that are absent come out empty in the transformed language.

The `translate-chars`, and `field-select` operations are specializations of the more general `translatematch` construct; RT includes these specializations as they correspond to commonly-used commands and they admit more precise and efficient modeling than the fully general case of `translate-match`.

To illustrate the operations concretely, Tab. 1 lists example command invocations and corresponding command types demonstrating each operation. For instance, `sed`'s transformation corresponds to `translate-match`, and `tr` uses `translate-chars`.

The behavior of `cut` has three cases: (1) if the input line lacks the delimiter entirely, it outputs the whole line, (2) if the input line includes the delimiter but fewer fields than those requested, it outputs the empty string for the missing fields, and (3) otherwise it outputs the requested fields separated by the delimiter and in ascending order of index. To capture this complex behavior, `cut`'s type performs field selection according to the stricter semantics of `field-select` for the sublanguage of inputs that contains the delimiter, and then

Tab. 1: Soundness and precision of operations. Each row describes a regular language operation, whether RT can compute the output language soundly and precisely, and an example command that is described with the operation. The last column describes the type of FST used to compute the output language, which can be a functional non-deterministic FST (FN-FST), a deterministic FST (DFST), or a non-deterministic FST (NFST).

Operation	Instance	*Sound?*	*Precise?*	Example (`Command` :: `Output Type`, for input language T)	FST Type
`translate-match`	String to string	✓	✓	`sed 's/old/new/'` :: `translate-match(`T`,old,new)`	FN-FST
	Anc. regex	✓	✓	`sed 's/^a.a*b/new/'` :: `translate-match(`T`,^a.a*b,new)`	FN-FST
	General case	✓	-	`sed 's/a.*/b/g'` :: `translate-match(`T`,a.*,b,true)`	NFST
`line-extract`	Cap. string	✓	✓	`grep -o 'pattern'` :: `line-extract(`T`,pattern)`	FN-FST
	Cap. anc. regex	✓	✓	`rg '^a.a*b'` :: `line-extract(`T`,^a.a*b)`	NFST
	Cap. regex	✓	-	`awk '/a.a*b/'` :: `line-extract(`T`,a.a*b)`	NFST
`field-select`	General case	✓	✓	`cut -d ' ' -f 2` :: `field-select(`T`&(.*␣.*),␣,2)|`T`&[^␣]*`	DFST
`translate-chars`	General case	✓	✓	`tr a-z A-Z` :: `translate-chars(`T`,a-z,A-Z)`	DFST

unions with the language of inputs that do not contain the delimiter.

While some common commands map directly to one operation, others compose multiple. For example, the type of `awk '{print $2}'` in RT's database is:

```
∀α.α →
  field-select(
    translate-chars(
      translate-match(α,"^[␣\t]+", ""),
    "␣,\t", "␣", squeeze=true),
  "␣", 2)
```

4.2 Finite State Transducers

RT implements the regular language operations shown in Fig. 5 by constructing finite state transducers (FSTs) and composing them the input automata representing regular types. An FST is a finite-state machine that not only recognizes input strings but also emits output strings along transitions. [40] This allows RT to define type-level string transformations that preserve regularity: the output of an FST applied to a regular language is itself regular.

Each new regular language operation that RT supports corresponds to its own FST construction. Once constructed, the FST is composed with the input automaton (corresponding to the input stream regular type) to produce a new automaton representing the transformed language (the output regular type). These constructions do not complicate typechecking: they essentially contribute one new step to `InferOutputType` in Alg. 1, which computes the result of any operations; hence, type checks remain in terms of simple regular languages.

Different operations require different classes of FSTs depending on their complexity and the structure of their pattern matching logic. Tab. 1 summarizes the operations supported by RT, including whether their output languages are computed soundly and precisely, and the class of FST used. Deterministic FSTs (DFSTs) are transducers with exactly one possible transition per input symbol in each state. They are used for transformations that are positionally local and unambiguous. Functional non-deterministic FSTs (FN-FSTs) allow multiple accepting paths for a given input, but all such paths produce the same output. This enables more expressive transformations while preserving precision. General non-deterministic FSTs (NFSTs) may produce multiple outputs for the same input and are used for the most general forms of pattern-based transformations. These transducers remain sound but may sacrifice precision—meaning the result of computing the corresponding operation may be a larger language than necessary.

Precision of transformation: Not all operations can be implemented perfectly precisely. Some instantiations of the operations are not regular transformations: consider the invocation `sed -E "s/(.*)/\1\1/"`, corresponding to `translate-match(A, (.*), \1\1)`, which duplicates input strings. The image of input language `.*` under this duplication operation is not a regular language [27]. On the other hand, other instantiations do perform regular transformations, but the transformations do not admit precise modeling. RT supports sound approximations in all such cases, even while supporting precise modeling for simple common cases; Tab. 1 breaks down which cases RT can and cannot support precisely.

RT precisely computes the output language for operations `field-select` and `translate-chars`. They are implemented using deterministic FSTs, which contain no ambiguous transitions. However, `line-extract` and `translate-match` operations are challenging to implement because determining exact match positions often requires nondeterministic guesses. Specific cases of `translate-match`—string matching and anchored patterns—are addressed with special transitions in the FST that recover from failed partial matches, which results in functional non-deterministic FSTs. Similarly, for the precise cases of `line-extract`, such as capturing an anchored regex pattern, RT relies on non-deterministic FSTs. In the general cases of

Tab. 2: RT heuristics. Each row describes a heuristic that RT uses to identify likely mistakes in command compositions, its description, and an example command composition that triggers the heuristic.

Name	Description	Example
Empty output	Command has empty output language	`cat data.txt | tr a b | grep pattern`
Ignored input	Command that expects no input	`cat data.txt | grep -rHn "word1" $list`
Useless composition	Stage does nothing	`echo "http://www.google.com" | cut -f3`
Lexicographic number sort	Sort numbers with non-numeric `sort`	`cat data.txt | sort | uniq -c | sort -r`

`line-extract` and `translate-match`, however, RT constructs non-deterministic FSTs that over-approximate the actual output language. These non-deterministic structures ensure soundness by covering all possible outputs but sacrifice precision. For instance, RT computes `translate-match("abab", "a.*b", c, global=true)` as `c?c`, an over-approximation of the precise output `c`. If a downstream command expects strictly `c`, this can lead to false positives.

5 Configurability and Concretization

RT can perform environment-based concretization, inferring input languages based on the contents of input files at the time of analysis, to provide more precise information (§5.1). Two types of annotations provide a way for developers to describe the specific shape of unknown inputs, or assert expectations about the language of desired outputs §5.2.

Also configurably, RT can warn about composition mistakes that do not strictly violate any command's input constraints, but rather behave in ways that are almost-always undesirable. For instance, a common bug is a pipeline that never produces output due to incorrect command composition. Using regular language type information, RT can warn developers about such mistakes ahead-of-time based on a set of heuristics about typical input and output languages (§5.3).

5.1 Environment-based Concretization

RT can be configured to leverage information from the environment during analysis to obtain information about actual inputs to reason precisely about program behavior within the local system context. For instance, by analyzing the content of an input file, RT can infer a regular language type capturing the current contents, and then use that type to be able to notify developers that a program will certainly be problematic if executed using that particular file. This can help developers stay ahead of typical mistakes that slow down development, such as using `cut` to select fields but forgetting to specify the field delimiter.

RT leverages concretization as part of a secondary specialization pass when type-checking programs. After performing normal type checking (*i.e.*, making no assumptions refining the range of possible values) and reporting any general issues with a program, RT then attempts concretization to offer finer-grained information. Specifically, if there are concrete inputs that are accessible at analysis-time on the local machine, including files or environment variables, RT reads the input and uses the entire contents verbatim as the input language in the same way as assumption annotations.

5.2 Annotations

Annotations let developers configure how RT checks command compositions, enabling more precise validation. While some compositions may seem unsafe for arbitrary inputs, they can be valid for specific input formats. For example, using `grep` to filter out `NA` lines before passing data to a command expecting numbers is safe when the input contains only numbers or `NA`. By annotating such expectations, developers help RT suppress irrelevant warnings and provide more accurate feedback.

Annotations also help developers use RT to catch bugs beyond unsafe compositions. For instance, a program's output may need to meet specific correctness properties—like not containing newlines before being sent over the network. With output assertions, developers can encode such expectations, allowing RT to verify them.

Developers can provide annotations using comments in a script's source, such as the following:

```
# @assume "cat $1" --> "[0-9]\t[A-Za-z]"
cat $1 | tr A-Z a-z | cut -f 2 | ...
```

In general, annotations appear as comments that begin with `@` and a keyword identifying the kind of annotation. Corresponding to the different kinds of information developers can offer, there are five distinct kinds of annotations. The first two are assumptions that RT should use while reasoning instead of computing the corresponding information: `assume` assumes the type of an invocation's output, and `input` assumes the type of input to the entire program. The remaining three are assertions that RT checks in addition to confirming input-output type compatibility: `assert` checks that the computed output type of an invocation is a subset of the annotated type, `expect` checks that the input an invocation it receives is a subset of the annotated type, and `output` checks that the program's output is a subset of the annotated type.

To help developers write annotations when necessary, RT supports human-friendly definitions for a variety of types that

desugar behind the scenes. These definitions capture common patterns such as ip-addresses, numbers in a floating-point decimal format, urls, and more (appendix A lists the full set of definitions RT supports, which is also extendable).

RT incorporates annotations into the type-checking process at different points for each kind of annotation. For assumption annotations, RT simply replaces the corresponding type with the assumed one at the relevant stage of the pipeline. For assertions, RT first computes the actual regular language type at the relevant stage, and then checks if that actual type is included in the asserted type—if so, it proceeds with the computed type, and otherwise raises an error reporting the failure.

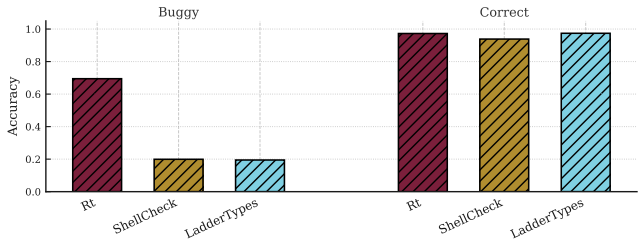


Fig. 6: Accuracy. The chart shows different accuracies of RT, ShellCheck, and LADDERTYPES in buggy script categories and correct script categories. Each category has three bars, from left to right: RT, ShellCheck, and LADDERTYPES.

5.3 Heuristics

RT includes several heuristics identifying compositions that do not strictly violate any command's input constraints, and hence are well-typed, but correspond to likely mistakes. That is, the circumstances identified by the heuristics are not necessarily problematic, but they capture common mistakes and undesirable behavior.

Tab. 2 lists all of the four heuristics RT considers, each of which describes a criteria for rejecting a program. The first heuristic is command invocations in pipelines that have an empty output language—therefore are guaranteed to have no output. The second is providing input to commands that expect none, such as `echo`. The third is pipeline stages that are guaranteed to have no effect on the pipeline contents. The fourth is using the `sort` command in lexicographic mode (without `-n`) on numeric inputs. Critically, the heuristics describe properties of compositions that depend on regular language input and output information. Hence, the heuristics capture common mistake patterns at a more semantic level than syntactic matching, which is not robust to syntactic variation.

6 Evaluation

We evaluate RT on hundreds of real-world programs, investigating the following questions: **Q1**) What is the effectiveness of RT? and **Q2**) What is the analysis performance of RT?

Overview of Results: RT's overall accuracy across all programs is 90%. Fig. 6 breaks this down into accuracy on buggy programs (71%) and correct ones (97%)—and when programs include extra type annotations to inform RT, its accuracy rises to 92% and 98% respectively. In comparison to state-of-the-art systems, RT is 40% more accurate than LADDERTYPES, and 40% more accurate than ShellCheck on buggy programs. Furthermore, RT provides confidence information indicating whether approximations may lead it to warn about a program, and for programs with definite confidence RT exhibits 100% accuracy (and 86% accuracy for possible-bug reports). RT analyzes every program in the set in <1s (avg. 0.02s).

6.1 Benchmark Suite

We use a comprehensive suite of both correct and buggy programs to answer these research questions. Since no established suite meeting both of these criteria already exists, we have collected and curated a new set of benchmarks.

Tab. 3 summarizes our benchmark suite, consisting of 7 different evaluation sets drawn from a variety of sources. The first three sets are drawn directly from the literature and community. The first set is examples of both buggy and correct programs from Sippel and Schirmeier's pipeline typing system, which we refer to as LADDERTYPES [52] (c.f. §7); second is all scripts in the Koala benchmark suite [33]; and third is the full set of programs from the Intercode ALPHA evaluation suite for natural-language to code synthesis systems [58]. These three sets come with explicit and implicit labels; some of the LADDERTYPES programs are provided as buggy examples, and all of the other programs are intended to be correct. However, in the process of collecting these sets, we identified two other bugs in the LADDERTYPES programs, three in the Koala benchmarks, and 17 in the Intercode suite, and therefore labeled these particular programs buggy.

The remaining benchmark sets focus on buggy pipelines from four sources. (1) LLM-generated programs, prompted to produce common shell mistakes. (2) Handwritten tests targeting specific composition patterns. (3) StackOverflow bugs identified via manual search and inspection. (4) GitHub scripts, extracted from bugfix commits.

We collected the GitHub programs using a partially automated process: first, we selected the top twenty repositories with shell code on GitHub (according to number of stars) using the GitHub query API; for each such repository, we filtered the history of commits (total 47080) to those which (1) modify a file ending with `.sh|.bash|.zsh` and (2) the modified line contains the pipe character `|`; for every such commit (total 1028), we prompt gpt-4o-mini to identify which

Tab. 3: Benchmark summary. #Correct and #Buggy columns show the number of programs that are correct or include one or more composition mistakes, respectively.

Set	#Correct	#Buggy	Description	Source
GitHub	57	57	Buggy and correct programs from bugfix commits	GitHub
StackOverflow	–	11	Buggy programs from StackOverflow	StackOverflow
LADDERTYPES	4	8	Example programs from the paper	[52]
Koala benchmarks	478	3	Programs from the Koala benchmark suite	[33]
Intercode ALPHA	188	17	Programs created for the evaluation of NL synthesis	[58]
LLM	–	120	Buggy programs generated by an LLM	GPT-4o [44]
Handwritten	3	8	Handwritten microbenchmarks	this paper
Total	730	224		

commits fix a bug relating to the contents of a pipeline; we manually inspected the resulting 328 commits to check their relevance, discarding commits that do not fix bugs, or for which the bug is unrelated to pipeline contents—ending up with 57 relevant commits. Each such commit provides two programs in our suite: the pre-commit script, which contains a bug (included in the buggy set), and the post-commit script, which fixes the bug (included in the correct set).

We collected the StackOverflow scripts manually, using both web search (with phrases such as "shell", "bash", and "pipeline"), and the StackExchange API, filtering for questions containing the keywords "error", "bug", "pipeline", or "unexpected" in the core site and Unix & Linux Stack Exchange sub-domain.[2] We manually reviewed the results and included only buggy programs, as these posts rarely provided corrected versions.

6.2 Q1: Effectiveness

To answer Q1, we calculate RT's accuracy in identifying composition mistakes across all benchmark sets. Accuracy is defined as the proportion of correct categorizations RT makes relative to the whole population; a system that always flags buggy programs and never correct ones is 100% accurate. We evaluate RT configured with its included type database—which covers 71/106 of the GNU core util commands and 86% of the command invocations in the GitHub set (*Cf.*§3.1).

Big picture: At a high level, RT has an overall accuracy of 90% (99 false positive/negatives), and identifies 150 bugs that state-of-the-art systems fail to detect. Fig. 6 breaks this down into accuracy on buggy programs (71%) and correct ones (97%)—and for programs with extra type annotations, its accuracy rises to 92 % and 98% respectively. Breaking RT's judgments down into definite and possibly-buggy warning levels, RT's definite warnings are 100% accurate, while its possibly-buggy warnings are 86%—the drop in accuracy reflecting the use of approximations and heuristics in its reasoning. Comparing the accuracy of RT, ShellCheck, and LADDERTYPES: RT significantly outperforms both baseline systems on buggy programs (by 40%) while maintaining comparable accuracy on correct programs.

[2] https://stackoverflow.com, https://unix.stackexchange.com

Tab. 4: Breakdown of configurations. Each column represents a different configuration of RT. The leftmost is RT with all features enabled, and following columns disable one feature: without heuristics, without FSTs, and without concretization. Rows indicate RT's true result under each configuration on correct and buggy scripts.

		RT	w/o heus	w/o FSTs	w/o con
w/o Ann.	Correct	697	707	579	696
	Buggy	158	85	176	158
	Acc	90%	83%	79%	90%
w/ Ann.	Correct	714	724	593	706
	Buggy	205	142	193	205
	Acc	96%	91%	82%	95%

6.2.1 Q1: Detailed Analysis

Tab. 4 breaks down RT's accuracy across correct and buggy benchmark subsets, reporting true positives and negatives based on whether RT correctly signals warnings for buggy programs or refrains from warning on correct ones. The leftmost column reports results for the full configuration of RT. The remaining columns list results for a version of RT with one or more features disabled, thereby quantifying the benefits offered by each feature in isolation or combination. Overall, RT has 90% accuracy in our evaluation suite, with 33 false positives on correct scripts and 66 false negatives on incorrect ones. With annotations, RT has 96% accuracy on the programs, with 26 false positives on correct scripts and 19 false negatives on incorrect ones.

Sources of inaccuracy: To better understand RT's inaccurate results, Tab. 5a breaks down their causes for the maximal configuration. RT produces 10 false positives due to programs that violate heuristics, but are actually correct (c.f. §5.3). RT

Tab. 5: Breakdown of false results. The tables break down the false results for the full configuration of RT evaluated on the benchmark set with and without annotations.

(a) Without annotations

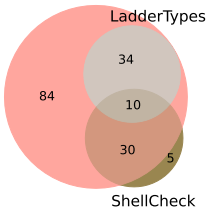
Category	Group	#	Description	Example
False Positive	Bad heuristics	10	Correct scripts flagged by heuristic	`du -ah /workspace | sort -rh`
	Over-approximation	23	Imprecise command type	`cat ${input} | sort -n`
False Negative	Unexpressible semantic	19	Contents are well-typed, but *e.g.*, unsorted	`... | sort | cut -d',' -f10 | uniq | ...`
	Wrong output format	47	Expected output format is not provided	`echo "$INDEX" | grep '^ D '`

(b) With annotations

Category	Group	#	Description	Example
False Positive	Bad heuristics	10	Correct scripts flagged by heuristic	`du -ah /workspace | sort -rh`
	Over-approximation	6	Imprecise command type	`... | sed -E 's/(.*)/\1\1/'`
False Negative	Unexpressible semantic	19	Contents are well-typed, but *e.g.*, unsorted	`... | sort | cut -d',' -f10 | uniq | ...`

reports 23 false positives due to over-approximations in type inference. For instance, the file referenced by `${input}` in the pipeline `cat ${input} | sort -n` is unknown ahead of time. Consequently, RT conservatively infers the output of `cat` as `.*`, which is sound but imprecise. RT produces 19 false negatives for programs that are well-typed but the semantics of the programs are incorrect according to the program's original source (*i.e.*, the bug is not of a nature that RT could detect). RT produces 47 false negatives due to the absence of expected output specifications. Without user-provided assertion annotations to define the required format, the script evaluates as well-typed. Tab. 5b shows the false results for the full configuration of RT evaluated on the benchmark set with annotations. RT eliminates all false negatives caused by the absence of expected output specifications with annotations. However, RT still produces 6 false positives related to over-approximations. These are invocations that RT models soundly but imprecisely due to inherent complexity of the commands—such as complex uses of `awk`, `sed`. Properly supporting all possible invocations of these commands demands a specialized language analysis (in the case of `awk`); `sed` invocations with complex capture groups, however, pose fundamental challenges (§4.2).

Comparison with state-of-the-art systems: We compare RT's accuracy to that of LADDERTYPES and ShellCheck. Like RT, LADDERTYPES classifies programs as buggy or correct. However, LADDERTYPES comes by default with a much smaller set of type declarations than RT, so to enable fair comparison we manually add reasonable versions of all RT's simple type declarations, and add a fallback type declaration mirroring RT's. ShellCheck differs from RT and LADDERTYPES in that it emits a wide range of warnings—many irrelevant to our setting. To bring it into parity for comparison, we consider a program buggy under ShellCheck if it produces any warning related to pipelines or output stream contents,[3] and correct otherwise.



The diagram on the right compares the three systems' ability to detect bugs in our evaluation suite, showing per-program results. The RT Euler diagram illustrates the number of buggy programs each system identifies, with overlaps indicating agreement across systems. To ensure a fair comparison, RT is evaluated without annotations, as neither ShellCheck nor LADDERTYPES benefits from developer-supplied information. RT uniquely detects 84 bugs, compared to 5 unique to ShellCheck and 0 to LADDERTYPES. Notably, 61 bugs remain undetected by all three systems in the absence of annotations.

Digging into the nature of the buggy programs that ShellCheck identifies but RT does not reveals that all but one are generic warnings that are not relevant for the particular bug at hand. The one warning that is relevant identifies an unsafe use of `ls -l ... | xargs rm`. Hence, ShellCheck identifies the composition bug only once.

6.3 Q2: Analysis Performance

Experimental setup: All evaluations were performed on a single machine running Ubuntu 20.04, equipped with an AMD Ryzen 7 4800H processor (2.9 GHz, 8-core) and 16 GB of RAM, using OpenJDK 17.02, Python 3.10, and the automaton package dk.brics.automaton (version 1.12-4).

Results: Tab. 6 shows the average evaluation time for RT,

[3] Ignored (irrelevant) warnings: SC2012,SC2046,SC2086,SC2018,SC2019, SC2002,SC2006,SC2009,SC2035,SC2060,SC2061,SC2062,SC2063,SC2126, SC2154,SC2185,SC2196,SC2225.

Tab. 6: Average time spent per program across RT, ShellCheck, and LADDERTYPES, in seconds.

System	Average	Min	Max
RT	0.0206	0.0003	0.9031
ShellCheck	0.0179	0.0100	0.0380
LADDERTYPES	3.0815	0.2340	14.8610

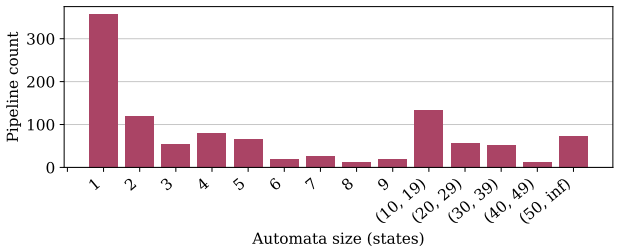


Fig. 7: Automata size. The histogram depicts counts of programs with maximum-size automata (number of states in the minimal DFA) in each bucket along the x-axis. Most automata have small numbers of states, so the bucketing granularity decreases for larger sizes.

ShellCheck, and LADDERTYPES. The evaluation time of LADDERTYPES is the longest, as it relies on an inefficient type lookup using an external shell script to match command invocations against predefined regular expressions in its type database. Indeed, all but one of the programs taking more than 0.2s to analyze are annotated with relatively complex, large input types. The longest time is for a 5-stage pipeline at 0.90s, which has an input annotation requiring a minimal DFA with 105 states.

RT's overall performance is dependent on the complexity of the types it reasons about, one proxy for which is the size of minimal automata representing the types. Fig. 7 shows the distribution of automata sizes RT constructs for the types in the evaluation suite, where each recorded automaton size corresponds to the largest DFA within a given program. Most programs contain minimal DFAs with fewer than 10 states. However, several programs include large DFAs; this occurs primarily when input or variable annotations involve lengthy regular expressions, resulting in complex DFAs even without additional transformation.

7 Related Work

Regular expressions for stream processing: Prior works [4,34] have considered the use of unambiguous regular expressions to describe the hierarchical decomposition of streams into finite windows for computing quantitative queries. The restriction of unambiguity is imposed to ensure that streaming queries have a uniquely determined output. In these works, regular expressions are not used to express correctness properties or data invariants for streams.

Types for POSIX shell pipelines: LadderTypes [52] models pipeline contents using types familiar from statically typed programming languages (*e.g.*, `Int` or `Sequence[Byte]`), encoding hierarchies of types (dubbed ladders) connected by a "represented as" relationship. Hence, it is a fundamentally different type system design, with a stricter notion of compatibility: two ladder types are compatible only if the full interpretation of a downstream consumer is included in the representation of the upstream producer's output.

Static analyses for the shell: Other kinds of static analyses have been proposed for the shell. ABash [36] analyzes scripts for expansion and word-splitting-related bugs, with a particular emphasis on code-injection style vulnerabilities. The CoLiS shell [29] includes a static analyzer that identifies dangerous or undesirable filesystem actions. Rodriguez and Wang [48] statically infer filesystem pre-conditions for file manipulation scripts. These analyses target different kinds of bugs and script behaviors than RT.

Semantic models of the shell: Semantic models of the shell provide critical underpinnings for understanding and reasoning about the semantics of shell scripts ahead of execution. Smoosh [22] defines the first mechanized formal semantics for the POSIX shell. The CoLiS language includes a formally verified interpreter [30], thereby defining the semantics of the (similar, but not POSIX shell) language. Such models inform how static analyses like RT should reason about shell programs, but are not analyses themselves.

Regular languages for static string reasoning: Regular languages provide a well-known formalism for reasoning about string-like values that have been employed in static analyses for other string-intensive contexts like jQuery [45] and security problems like code injection [5]. Rodriguez and Wang [48] use an automata-based representation of file paths, and leverage FSTs to manipulate them.

Shell replacements and shell-like languages or libraries: Both the literature and open-source community abound with shell-like scripting languages. These include traditional-looking shells with features drawn from typical general-purpose languages [10, 14]; libraries and embeddings in general purpose languages which maintain the essential design of byte-level communication streams [1, 7, 11, 16, 19, 21, 23, 49, 57]; languages that introduce higher level structure in at least some process communication [2, 25, 38, 50]; languages that also feature a type system that can catch some composition mistakes [26, 35, 51]; and languages that offer security monitoring in shell programming [41]. Unlike this direction of work, RT is a static analysis tool for the POSIX shell rather than a new language.

Linters for the shell and shell-wrapping systems: Several shell linting tools have been developed to analyze shell scripts. Checkbashisms [8] mainly focuses on identifying the

usage of shell syntax that is exclusive to the bash shell environment. PSScriptAnalyzer [37] specializes in detecting common syntax issues and code smells in PowerShell scripts, while ShellCheck [18] does the similar job for various popular shells like Bash and zsh. SecureCode [12] extracts shell code embedded in DevOps orchestration frameworks like Ansible in order to analyze them with ShellCheck. DRIVE [59] similarly extracts shell code from Dockerfiles and applies rule-based patterns to identify code-smells and possible bugs. However, these tools rely on syntax-based pattern matching, which limits their analysis to syntactic issues. In contrast, RT models stream contents, enabling semantic analysis beyond predefined syntactic rules.

Automated program repair for the shell: NoFAQ is a rule-based command repairing tool [17]. It matches the given command and error message with bugfix examples in a database and suggests the corresponding fixes. RT does not aim to suggest fixes, and identifies bugs semantically rather than with rules/patterns over syntax or error messages.

8 Conclusion

RT demonstrates that regular types provide an effective and efficient basis for statically detecting input-output composition mistakes in the UNIX shell. Regular types capture significant structure in stream contents while remaining faithful to the byte-oriented nature of shell communication, leverage the rich literature around regular languages and automata, and lean into the familiarity of regular expressions that are already ubiquitous in UNIX environments.

Upon acceptance, RT's implementation and all benchmarks presented in this paper, will be available as an MIT-licensed open-source project at:

https://github.com/atlas-brown/rt

References

[1] Eshell: The emacs shell, 2025. Accessed: 2025-03-22.

[2] Nushell: A new type of shell, 2025. Accessed: 2025-03-22.

[3] Pcre - perl compatible regular expressions. PCRE.org, 2025.

[4] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In Peter Thiemann, editor, *Proceedings of the 25th European Symposium on Programming (ESOP '16)*, volume 9632 of *Lecture Notes in Computer Science*, pages 15–40, Berlin, Heidelberg, 2016. Springer.

[5] Vincenzo Arceri and Isabella Mastroeni. Static program analysis for string manipulation languages. *Electronic Proceedings in Theoretical Computer Science*, 299:19–33, August 2019.

[6] Jon Bentley, Don Knuth, and Doug McIlroy. Programming pearls: a literate program. *Commun. ACM*, 29(6):471–483, June 1986.

[7] Joel Berger, Jaap Karssenberg, and R.L. Zwart. Zoidberg: A modular perl shell, 2025. Accessed: 2025-03-22.

[8] R Braakman, J Rodin, J Gilbey, and M Hobley. Checkbashisms. https://sourceforge.net/projects/checkbaskisms/, 2015. Accessed: 2024-11-26.

[9] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[10] Andy Chu. Oils: Our upgrade path from bash to a better language and runtime. https://oils.pub/, 2025. Accessed: 2025-03-22.

[11] David Crawshaw. Neugram: scripting language integrated with go, 2025. Accessed: 2025-03-22.

[12] Ting Dai, Alexei Karve, Grzegorz Koper, and Sai Zeng. Automatically detecting risky scripts in infrastructure code. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 358–371, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Peter Deutsch. Gzip file format specification version 4.3, May 1996. RFC 1952.

[14] Elvish Developers. Elvish shell. https://elv.sh/, 2025. Accessed: 2025-03-22.

[15] Mark Jason Dominus. Perl regular expression matching is np-hard. https://perl.plover.com/NPC/. Accessed: 2025-04-15.

[16] Dundalek. Closh: Bash-like shell based on clojure, 2025. Accessed: 2025-03-22.

[17] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. Nofaq: Synthesizing command repairs from examples. corr abs/1608.08219 (2016), 2016.

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

[19] Tomer Filiba. Plumbum: Shell combinators and more, 2025. Accessed: 2025-03-22.

[20] Inc. GitHub. The state of open source software. https://octoverse.github.com/#top-languages-over-the-years, 2024. Accessed: 2024-11-01.

[21] Gabriel Gonzalez. Turtle: Shell programming, haskell style. https://hackage.haskell.org/package/turtle, 2025. Accessed: 2025-03-22.

[22] Michael Greenberg and Austin J. Blatt. Executable formal semantics for the POSIX shell. *PACMPL*, 4(POPL):43:1–43:30, December 2019.

[23] Li Haoyi. Ammonite: Scala scripting, 2025. Accessed: 2025-03-22.

[24] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2 edition, 2016.

[25] William Gallard Hatch and Matthew Flatt. Rash: from reckless interactions to reliable programs. *ACM SIGPLAN Notices*, 53(9):28–39, 2018.

[26] Alec Heller and Jesse A. Tov. Caml-shcaml: an ocaml library for unix shell programming. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 79–90, New York, NY, USA, 2008. Association for Computing Machinery.

[27] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education, 3rd edition, 2006.

[28] Jeroen Janssens. *Data science at the command line*. O'Reilly Media, Sebastopol, CA, October 2014.

[29] Nicolas Jeannerod. *Verification of Shell Scripts Performing File Hierarchy Transformations*. PhD thesis, University of Paris, 2021.

[30] Nicolas Jeannerod, Claude Marché, and Ralf Treinen. A Formally Verified Interpreter for a Shell-like Programming Language. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments*, volume 10712, Heidelberg, Germany, July 2017.

[31] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. Characterizing the security of github {CI} workflows. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2747–2763, 2022.

[32] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. 3 edition, 2003.

[33] Evangelos Lamprou, Ethan Williams, Georgios Kaoukis, Zhuoxuan Zhang, Michael Greenberg, Konstantinos Kallas, Lukas Lazarek, and Nikos Vasilakis. The koala benchmarks for the shell: Characterization and implications. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC '25)*, pages 449–64, Boston, MA, July 2025. USENIX Association.

[34] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '17, pages 693–708, New York, NY, USA, 2017. ACM.

[35] Kouji Matsui. A proposal for an interactive shell based on a typed lambda calculus, 2021.

[36] Karl Mazurak and Steve Zdancewic. ABash: Finding bugs in bash scripts. In *PLAS*, pages 105–114, San Diego California USA, June 2007. ACM.

[37] Microsoft. Psscriptanalyzer. https://github.com/PowerShell/PSScriptAnalyzer, 2020. Accessed: 2024-11-26.

[38] Microsoft. What is powershell?, 2025. Accessed: 2025-03-22.

[39] J.C. Mitchell. *Foundations for Programming Languages*. Foundations of computing. MIT Press, 1996.

[40] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

[41] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. {SHILL}: A secure shell scripting language. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 183–199, 2014.

[42] Jason Morris, Chris McCubbin, and Raymond Page. *Hands-On Data Science with the Command Line*. Packt Publishing, Birmingham, England, January 2019.

[43] Shaun Nichols. Scary code of the week: Steam cleans linux pcs. *The Register*, January 2015. Accessed: 2025-03-06.

[44] OpenAI. Gpt-4o system card, 2024.

[45] Changhee Park, Hyeonseung Im, and Sukyoung Ryu. Precise and scalable static analysis of jquery using a regular expression domain. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 25–36, New York, NY, USA, 2016. Association for Computing Machinery.

[46] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[47] Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia. POSH: A Data-Aware shell. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 617–631. USENIX Association, July 2020.

[48] Rodney Rodriguez and Xiaoyin Wang. Understanding execution environment of file-manipulation scripts by extracting pre-conditions. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 406–410. IEEE, 2021.

[49] Anthony Scopatz. Xonsh: Python-powered shell, 2025. Accessed: 2025-03-22.

[50] Olin Shivers. Scsh manual 0.6.7. https://scsh.net/docu/html/man.html, 2006.

[51] Jon Shultis. A functional shell. In *Proceedings of the 1983 ACM SIGPLAN Symposium on Programming Language Issues in Software Systems*, SIGPLAN '83, pages 202–211, New York, NY, USA, 1983. Association for Computing Machinery.

[52] Michael Sippel and Horst Schirmeier. Process composition with typed unix pipes. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*, pages 34–40, 2023.

[53] Slashdot. itunes 2.0 installer deletes hard drives, 2001. Accessed: 2025-03-06.

[54] Valve Software. Moved /.local/share/steam. ran steam. it deleted everything on system owned by user., 2023. Accessed: 2023-10-04.

[55] The Open Group. Regular expressions – posix basic regular expressions. https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html, 2018. Accessed: April 15, 2025.

[56] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. Pash: light-touch data-parallel shell processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 49–66, New York, NY, USA, 2021. Association for Computing Machinery.

[57] Greg Weber, Petr Rockai, Andreas Abel, John Wiegley, Michael Snoyman, and psibi. Shelly: Shell-like (systems) programming in haskell, 2025. Accessed: 2025-03-22.

[58] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback, 2023.

[59] Yu Zhou, Weilin Zhan, Zi Li, Tingting Han, Taolue Chen, and Harald Gall. Drive: Dockerfile rule mining and violation detection. *ACM Trans. Softw. Eng. Methodol.*, 33(2), 2023.

A Syntactic Sugar Definitions

Syntactic Sugar	Regular Type
{ipv4}	((25[0-5]|2[0-4][0-9]|1?[0-9][0-9]?)\.)3(25[0-5]|2[0-4][0-9]|1?[0-9][0-9]?)
{integer}	[-+]?[0-9]+
{float}	[-+]?([0-9]+\.?[0-9]*|[0-9]*\.[0-9]+)
{hex}	0x[0-9a-fA-F]+
{percentage}	(100(\.0+)?|[0-9]{1,2}(\.[0-9]+)?)%
{currency_usd}	\$\s?[0-9]*.?[0-9]*
{mac_address}	([0-9a-fA-F]{2}[:-])5([0-9a-fA-F]{2})
{email_address}	[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}
{url}	https?:\/\/[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}[^{}]*
{iso_date}	[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])
{year}	[12][0-9]{3}
{month_name}	[Jj]an(uary)?|[Ff]eb(ruary)?|[Mm]ar(ch)?|[Aa]pr(il)?|[Mm]ay|[Jj]un(e)?|...
{day_of_week}	[Mm]on(day)?|[Tt]ue(sday)?|[Ww]ed(nesday)?|[Tt]hu(rsday)?|[Ff]ri(day)?|...
{time_24h}	([01]?[0-9]|2[0-3]):[0-5][0-9](:[0-5][0-9])?
{time_12h}	([1-9]|1[0-2])(:[0-5][0-9](:[0-5][0-9])?)?\s?[AP]M
{uuid}	[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}
{hex_color}	#(?:[0-9a-fA-F]{3}){1,2}\b
{semantic_version}	(0|[1-9]\d*)\.(0|[1-9]\d*)\.(0|[1-9]\d*)
{abs_path}	(/[^{}/]*)+/?
{file_extension}	\.[a-zA-Z0-9]+$
{hidden_file}	\.[a-zA-Z0-9_\-\.]+
{perms}	[-d]([r-][w-][x-]){3}
{html_tag}	<([a-z]+)([^{}<]+)*(?:>(.*)<\/\1>|\s+\/>)
{domain_name}	([a-zA-Z0-9]([a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?\.)+[a-zA-Z]{2,}
{subdomain}	([a-zA-Z0-9-]+\.)+
{url_slug}	[a-z0-9]+(?:-[a-z0-9]+)*
{ipv6}	([0-9a-fA-F]{1,4}:){7,7}[0-9a-fA-F]{1,4}|([0-9a-fA-F]{1,4}:){1,7}:|...
{us_phone}	(\+?1[-.]?)?\(?[0-9]{3}\)?[-.]?[0-9]{3}[-.]?[0-9]{4}
{base64}	(?:[A-Za-z0-9+/]{4})*(?:[A-Za-z0-9+/]{2}==|[A-Za-z0-9+/]{3}=)?
{md5}	[a-fA-F0-9]{32}
{sha1}	[a-fA-F0-9]{40}
{sha256}	[a-fA-F0-9]{64}
{subnet_mask}	(255\.){3}(0|128|192|224|240|248|252|254|255)
{username_unix}	[a-z_][a-z0-9_-]*

B FST Construction Example

To more concretely understand how RT constructs the FST for a given regular language operation, consider the `translate-chars` operation with `squeeze=true`, which transforms an input language `A` by collapsing consecutive occurrences of characters in the translated-to set into a single instance. Fig. 8 illustrates this process for the input language `(␣b␣)*`, with space as the character being squeezed. Starting from the DFA of the input (a) and the FST representing the `translate-chars` operation (b), RT computes their product (c) and derives an NFA over the FST's outputs (d), yielding the transformed language.

At a high level, RT computes the `translate-chars` operation over regular languages by constructing the FST for the particular bytes of interest, computing its product with the input language automata, and constructing a resulting NFA. The other operations all have similar constructions to compute their result languages; broadly, they each involve different combinations of NFA/FST construction and regular language/finite automata operations.

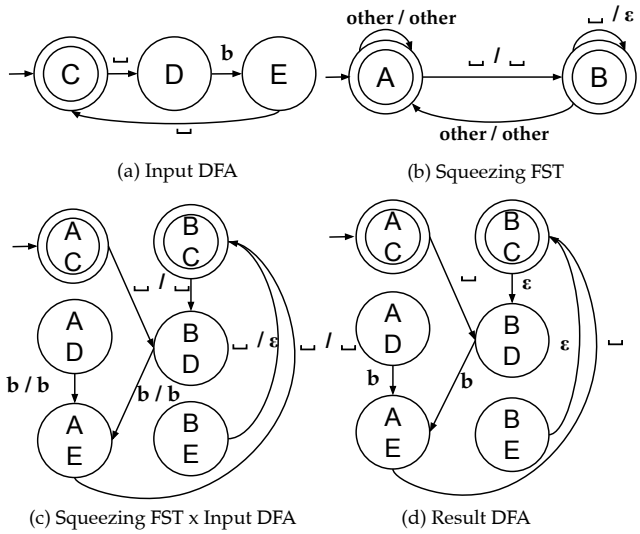


Fig. 8: FST construction example. RT construscts the FST `translate-chars((␣b␣)*, ␣, ␣, squeeze=true)` by constructing the DFA for the regular language `(␣b␣)*` (a) and then taking its product with the `translate-chars` FST (b). Taking the output of each transition of the resulting FST (c) produces the output DFA corresponding to the transformation of the input language (d).

C Extended Examples

The following examples illustrate cases of real-world shell programs, and how the RT type-system infers types for them. For each program, the first column describes the type that has been assigned by RT's built-in type database or one of the optional user annotations. The second column is the output type computed by the RT type-checking algorithm (alg. 1) for each pipeline stage.

NOAA weather data processing: The script in Fig. 9 downloads and processes weather data from the NOAA FTP server. It retrieves gzipped files, extracts each one, filters invalid lines, and sorts the data to find the maximum temperature for each year.

```
# - max_temp.sh -
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/;" |
  # .* -> <gz_fmt>*
  xargs -n 1 curl -s |
  # <gz_fmt>* -> .{88}[0-9]{4}*
  gunzip |
  cut -c 89-92 |
  grep -iv 999 |
  sort -rn |
  head -n 1 |
  sed "s/^/Max ($y): /"
done
```

Type	RT Computed Type
`y: 201[5-9]`	
`() → .*`	`.*`
`∀α.α → α & .*gz.*`	`.*gz.*`
`∀α.α →translate-chars(α, ␣, ␣, squeeze=true)`	`.*gz.* &!(.*␣␣.*)`
`∀α.α →field-select(α& (.*␣.*), ␣, 9)|α& [^␣]*`	`[^␣]*`
`∀α.α →translate-match(α, ^, $base/$y)`	`${base}/201[5-9][^␣]*`
`.* → gz_fmt`	`gz_fmt`
`gz_fmt* → .{88}[0-9]{4}.*`	`.{88}[0-9]{4}.*`
`∀α.α → field-select(α, (), 89-92)`	`[0-9]{4}`
`∀α.α → α & (.*999.*)`	`[0-9]{4} &!(.*999.*)`
`∀α ∈ [␣\t]*[-+]?[0-9]+.* . α → α`	`[0-9]{4} &!(.*999.*)`
`∀α.α → α`	`[0-9]{4} &!(.*999.*)`
`∀α.α → translate-match(α, ^, Max \($y\): )`	`Max \(201[5-9]\): [0-9]{4} & !(.*999.*)`

Fig. 9: NOAA weather data processing script. The script downloads and processes weather data from the NOAA FTP server, extracting the maximum temperature for each year. Here, `gz_fmt` is a type representing gzipped files [13].

Web crawler and indexer: This example involves a simple shell-based search engine which crawls a list of URLs, extracts the text content from each page, and indexes it. It comprises three main components: the engine (Fig. 10), the crawler (Fig. 11), and the indexer (Fig. 12), with three additional utilities that are used to normalize the text into a stream of terms (Fig. 13), calculate 1,2,3-grams from that stream (Fig. 14), and invert the n-grams into a partial index, where each n-gram is associated with a set of

URLs (Fig. 15). A query script is also provided to search the index for specific terms (Fig. 16).

```
# -- engine.sh --                    Type                    RT Computed Type
cd "$(dirname "$0")" || exit
cat /dev/null > d/visited.txt

# d/urls.txt: <url>|"stop"
while read -r url; do                 url: () → url | "stop"

  if [[ "$url" == "stop" ]]; then
    break;
  fi

  ./crawl.sh "$url" >d/content.txt    .* → .*                 ()
  ./index.sh d/content.txt "$url"     .* → .*                 .*

  # visited.txt: <url>*
  # urls.txt: <url>|stop+
  if  [[ "$(cat d/visited.txt | wc -l)" -ge \
         "$(cat d/urls.txt | wc -l)" ]]; then
        break;
  fi

done < <(tail -f d/urls.txt)
```

Fig. 10: Search engine entry point. The script reads URLs from a file, crawls each URL, and indexes the content. It stops when all URLs have been visited.

```
# - crawl.sh -                        Type                          RT Computed Type

# "$1": <url>
echo "$1" >>d/visited.txt

# getURLS.js: .* -> <url>
# getText.js: .* -> .*

curl -sL "$1" |                       () → .*                       .*
tee                                   ∀α.α → α                      .*
>(c/getURLs.js "$1" |                 .* → <url>                    <url>
  grep -vxf d/visited.txt             ∀α.α → α &![[visited.txt]]    <url> &!([[visited.txt]])
    >>d/urls.txt) |
c/getText.js                          .* → .*                       .*
```

Fig. 11: Web crawling script. The script fetches a webpage, extracts URLs, and saves the text content. After each iteration, it also updates the list of visited URLs.

```
# - index.sh -                        Type                    RT Computed Type

# "$1": <path>
# "$2": <url>

# def p-index: ([a-z]+)( [a-z]+)?( [a-z]+)? \| [0-9]+ \| <url>
# def index: ([a-z]+)( [a-z]+)?( [a-z]+)? \|(<url> [0-9]+ ?)*
# c/merge.js: <p-index> -> <index>
# c/stem.js: [a-z]+ -> [a-z]+

# d/global-index.txt: <index>

cat "$1" |                            () → .*
  c/process.sh |                      .* → [a-z]+             [a-z]+
  c/stem.js |                         [a-z]+ → [a-z]+         [a-z]+
  c/combine.sh |                      .* → .*                 ([a-z]+)|(([a-z]+)\t([a-z]+)?)|(([a-z]+)\t([a-z]+)?\t([a-z]+)?)
  c/invert.sh "$2" |                  .* → .*                 <p-index>
  c/merge.js d/global-index.txt |     <p-index> → <index>     <index>
  sort -o d/global-index.txt          ∀α.α → α                d/global-index: <index> stdout: ()
```

Fig. 12: Indexer. The script processes the content of the webpage pointed to by the given URL, extracts n-grams, and updates the global index with the new data.

# – c/process.sh –	Type	RT Computed Type
`tr -cs A-Za-z '\n' |`	`∀α.α → translate-chars(α, A-Za-z, \n, true)`	`[A-Za-z]+`
`tr '[:upper:]' '[:lower:]' |`	`∀α.α → translate-chars(α, [:upper:], [:lower:])`	`[a-z]+`
`grep -vwFf d/stopwords.txt`	`∀α.α → α &![[stopwords.txt]])`	`[a-z]+ &!([[stopwords.txt]])`

Fig. 13: Preprocessing. The script processes the input text (webpage content) by removing stopwords and converting it to lowercase.

# – c/combine.sh –	Type	RT Computed Type
`[ -p p1 ] || mkfifo p1`		
`[ -p p2 ] || mkfifo p2`		
`[ -p p3 ] || mkfifo p3`		
`tee`	`∀α.α → α`	`α`
`  >(sort) \`	`∀α.α → α`	`α`
`  >(tee p1 |`	`∀α.α → α`	`α`
`    tail +2 |`	`∀α.α → α`	`α`
`    paste p1 - |`	`∀α,β.α × β → α\tβ`	`α\tα`
`    sort)`	`∀α.α → α`	`α\tα`
`  >(tee p2 |`	`∀α.α → α`	`α`
`    tail +2 |`	`∀α.α → α`	`α`
`    paste p2 - |`	`∀α,β.α × β → α\tβ`	`α\tα`
`    tee p3 |`	`∀α.α → α`	`α\tα`
`    cut -f 1 |`	`∀α.α → field-select(α, \t, 1)`	`α`
`    tail +3 |`	`∀α.α → α`	`α`
`    paste p3 - |`	`∀α.β,α × β → α\tβ`	`α\tα\tα`
`    sort)`	`∀α.α → α`	`α\tα\tα`
`> /dev/null`		

Fig. 14: N-gram generator script. The script generates all 1,2,3-grams from an input stream of words.

# – c/invert.sh –	Type	RT Computed Type
`# "$1": <url>`		
`grep -vE '[[:space:]]+$' |`	`∀α.α → α &!(.*[[:space:]]+))`	`([a-z]+)(\t([a-z]+)){0,2}`
`sort |`	`∀α.α → α`	`([a-z]+)(\t([a-z]+)){0,2}`
`uniq -c |`	`∀α.α → ␣*[0-9]+ α`	`␣*t[0-9]+ ([a-z]+)(\t([a-z]+)){0,2}`
`# .* -> <p-index>`		
`awk '{print $2,$3,$4,"|",$1,"|"}' |`	`.* → <p-index>`	`([a-z]+)( [a-z]+)?( [a-z]+)? | [0-9]+ |`
`sed 's/\s\+/ /g' |`	`∀α.α → translate-chars(α, ␣+, ␣, true)`	`([a-z]+)( [a-z]+)?( [a-z]+)? | [0-9]+ |`
`sort |`	`∀α.α → α`	`([a-z]+)( [a-z]+)?( [a-z]+)? | [0-9]+ |`
`sed "s|$| $1|"`	`∀α.α → translate-match(α, $, ␣$1)`	`([a-z]+)( [a-z]+)?( [a-z]+)? | [0-9]+ | <url>`

Fig. 15: Inverter script. The script inverts the n-grams to create a partial inverted index, associating each n-gram with its frequency and the URL it came from. Here, the value of `stdin` is a stream of n-grams, and the value of `$1` is the URL.

# – query.sh –	Type	RT Computed Type
`# c/stem.js: [a-z]+ -> [a-z]+`		
`grep $(`	`∀α.α → α &.*$(...).*`	`([a-z]+)( [a-z]+)?( [a-z]+)? |(<url> [0-9]+ ?)* &.*[a-z]+.*`
`  echo "$1" |`	`() → [a-z]+`	`[a-z]+`
`  ./c/process.sh |`	`.* → [a-z]+`	`[a-z]+`
`  ./c/stem.js |`	`[a-z]+ → [a-z]+`	`[a-z]+`
`  tr -d "\r\n"`	`∀α.α → translate-chars(α, \n\r, "")`	`[a-z]+`
`) d/global-index.txt`	`.*`	`d/global-index: ([a-z]+)( [a-z]+)?( [a-z]+)? |(<url> [0-9]+ ?)*`

Fig. 16: Search engine query script. The script processes the input query (a string of 1-3 words), stems it, and searches the global index for matching entries.

```
mkfifo t{0..14}
# PaSh DFG1: start
tr A-Z a-z < f1.md > t0 &
tr A-Z a-z < f2.md > t1 &
tr -d[:punct:] < t0 > t2 &
tr -d[:punct:] < t1 > t3 &
sort < t2 > t4 &
sort < t3 > t5 &
sort -m t4 t5 > t6 &
split t7 t8 < t6 &
# ...
uniq < t7 > t11 &
uniq < t8 > t12 &
# ...
tee t9 > t10 < dict.txt &
grep -vx -f t9 - < t11 > t13 &
grep -vx -f t10 - < t12 > t14 &
cat t13 t14 > out &
wait
rm t{0..14}
# PaSh DFG1: end
mkfifo t{0..8}
# PaSh DFG2: start
split t0 t1 < out &
wc -l < t0 > t2 &
wc -l < t1 > t3 &
paste -d+ t2 t3
| bc > t4 &
  split t5 t6 < t4 &
sed 's/$/ typos!/' < t5 > t7 &
sed 's/$/ typos!/' < t6 > t8 &
cat t7 t8 &
wait
rm t{0..8}
# PaSh DFG2: end
```

Type	RT Computed Type
$\forall\alpha.\alpha \to$ `translate-chars(`α`, A-Z, a-z)`	`[^A-Z]*`
$\forall\alpha.\alpha \to$ `translate-chars(`α`, A-Z, a-z)`	`[^A-Z]*`
$\forall\alpha.\alpha \to$ `translate-chars(`α`, [:punct:], "")`	`[^A-Z[:punct:]]*`
$\forall\alpha.\alpha \to$ `translate-chars(`α`, [:punct:], "")`	`[^A-Z[:punct:]]*`
$\forall\alpha.\alpha \to \alpha$	`[^A-Z[:punct:]]*`
$\forall\alpha.\alpha \to \alpha$	`[^A-Z[:punct:]]*`
$\forall\alpha.\alpha \to \alpha$	`[^A-Z[:punct:]]*`
$\forall\alpha.\alpha \to \alpha$	`[^A-Z[:punct:]]*`
$\forall\alpha.\alpha \to \alpha$	`[^A-Z[:punct:]]*`
$\forall\alpha.\alpha \to \alpha$	`[^A-Z[:punct:]]*`
$\forall\alpha.\alpha \to \alpha$	`[[dict.txt]]`
$\forall\alpha.\alpha \to \alpha$ `&!([[dict.txt]])`	`[^A-Z[:punct:]]* &!([[dict.txt]])`
$\forall\alpha.\alpha \to \alpha$ `&!([[dict.txt]])`	`[^A-Z[:punct:]]* &!([[dict.txt]])`
$\forall\alpha.\alpha \to \alpha$	`[^A-Z[:punct:]]* &!([[dict.txt]])`
$\forall\alpha.\alpha \to \alpha$	`[^A-Z[:punct:]]* &!([[dict.txt]])`
`.*` $\to$ `[0-9]+`	`[0-9]+`
`.*` $\to$ `[0-9]+`	`[0-9]+`
$\forall\alpha,\beta.\alpha \times \beta \to$ `\+`	`[0-9]+\+[0-9]+`
`.*` $\to$ `[0-9]+`	`[0-9]+`
$\forall\alpha.\alpha \to \alpha$	`[0-9]+`
$\forall\alpha.\alpha \to$ `translate-match(`α`, $, typos!)`	`[0-9]+ typos!`
$\forall\alpha.\alpha \to$ `translate-match(`α`, $, typos!)`	`[0-9]+ typos!`
$\forall\alpha.\alpha \to \alpha$	`[0-9]+ typos!`

Fig. 17: Parallelized spell checking script. This script is equivalent to the serial spell checker (Fig. 18), but it uses parallel processing to speed up the spell checking process. This script was produced by PaSh [56], using `./pa.sh --width=2`.

Spell checking (and parallel equivalent): The script in Fig. 18 checks the spelling of words in two files, `f1.md` and `f2.md`, against a dictionary file `dict.txt`. It outputs the number of misspelled words. The same script is then transformed using PaSh [56] to run on multiple parallel cores (Fig. 17).

```
# - spell.sh -
cat f1.md f2.md |
 tr A-Z a-z |
 tr -cs A-Za-z '\n' |
 sort |
 uniq |
 grep -vx -f dict.txt -
 > out

cat out |
 wc -l |
 sed 's/$/ typos!/'
```

Type	RT Computed Type
$() \to$ `.*`	`.*`
$\forall\alpha.\alpha \to$ `translate-chars(`α`, A-Z, a-z)`	`[^A-Z]*`
$\forall\alpha.\alpha \to$ `translate-chars(`α`, A-Za-z, \n, true, true)`	`[a-z]*`
$\forall\alpha.\alpha \to \alpha$	`[a-z]*`
$\forall\alpha.\alpha \to \alpha$	`[a-z]*`
$\forall\alpha.\alpha \to \alpha$ `&![[dict.txt]])`	`[a-z]* &!([[dict.txt]])`
$\forall\alpha.\alpha \to \alpha$	`[a-z]* &!([[dict.txt]])`
`.*` $\to$ `[0-9]+`	`[0-9]+`
$\forall\alpha.\alpha \to$ `translate-match(`α`, $, typos!)`	`[0-9]+ typos!`

Fig. 18: Spell checking script. The script reads two files, converts them to lowercase, removes punctuation, sorts the words, and checks them against a dictionary. It outputs the number of misspelled words.