Turing Completeness of GNU `find`: From `mkdir`-Assisted Loops to Standalone Computation

Keigo Oka
Google, Tokyo, Japan

Abstract

The Unix command `find` is among the first commands taught to beginners, yet remains indispensable for experienced engineers. In this paper, we demonstrate that `find` possesses unexpected computational power, establishing three Turing completeness results using the GNU implementation (a standard in Linux distributions). (1) `find` + `mkdir` is Turing complete. By encoding computational states as directory paths and using regex back-references to copy substrings, we simulate 2-tag systems using only the `find` and `mkdir` executables. (2) GNU `find` 4.9.0+ alone is Turing complete: by reading and writing to files during traversal, we simulate a two-counter machine without `mkdir`. (3) `find` + `mkdir` without regex back-references is still Turing complete: by a trick of encoding regex patterns directly into directory names, we achieve the same power.

These results place `find` among the "surprisingly Turing-complete" systems, highlighting the hidden complexity within seemingly simple standard utilities.

2012 ACM Subject Classification Software and its engineering → Command and control languages; Theory of computation → Computability

Keywords and phrases Turing completeness, GNU find, tag system, counter machine

Digital Object Identifier 10.4230/LIPIcs.FUN.2026.36

Related Version *ArXiv Version*: `https://arxiv.org/abs/2602.20762`

Supplementary Material *Software (Source Code)*: `https://github.com/ogiekako/pub_find_mkdir` archived at `swh:1:dir:e33cf38786da2c414b41c2fab2728a401c5fda32`

Funding *Keigo Oka*: Work done in a personal capacity.

1 Introduction

The Unix command `find` is among the first utilities introduced to students and system administrators. While `find` is designed to search for files in a directory hierarchy, its recursive traversal with the ability to execute commands opens the door to complex behaviors. In this paper, we demonstrate that `find` possesses unexpected computational power.[1]

Foundational work by Minsky [8] and Post [12] established that very simple systems, such as tag systems and counter machines, are capable of universal computation. This insight explains why "accidental" Turing completeness is a recurring theme in computer science: systems designed for specific, limited purposes often turn out to have enough power to perform arbitrary computations. Famous examples include the stream editor `sed` [3, 2], the x86 `mov` instruction [5], and even the rules of *Magic: The Gathering* [4]. In security research, such emergent computational power has been termed "weird machines" [1].

[1] A preliminary version of this work appeared on the author's blog [11] and attracted considerable interest [7].

1.1 Contributions

In this paper, we establish three Turing completeness results. Given any instance of a known universal computational model, we show how to construct a fixed sequence of commands (an *execution* in the sense of Definition 1) whose behavior faithfully simulates that instance. Conversely, execution of any sequence of `find` and `mkdir` commands on a file system can clearly be simulated by a Turing machine, so we obtain Turing equivalence; however, we focus on the more surprising direction.

1. **GNU `find` + `mkdir` is Turing complete** (Section 3): By encoding computational states as directory paths and using regex back-references to copy substrings, we simulate 2-tag systems. We use options added in GNU `find` 4.2.12, which was released in 2005.
2. **GNU `find` alone is Turing complete** (Section 4): By reading and writing to files during traversal, we simulate a two-counter machine without `mkdir`. This construction utilizes the `-files0-from` option added in GNU `find` 4.9.0, which was released in 2022.
3. **GNU `find` + `mkdir` without regex back-references is Turing complete** (Section 5 (sketch), Appendix A): By a technique of encoding regex patterns directly into directory names, we achieve the same power even without regex back-references.

Our constructions use $O(1)$-length path components (hence do not assume an increased `NAME_MAX`). In practice, resource limits such as storage capacity, maximum path length (`PATH_MAX`, typically 4096 bytes), and finite inodes cap the number of simulation steps that can be realized on a given machine.

These results place `find` among the "surprisingly Turing-complete" systems, highlighting the hidden complexity within seemingly simple standard utilities.

We implement our constructions and confirm them on toy examples (Appendix B).

1.2 Applicability and Standards

We restrict our attention to GNU `find`, because (1) it is standard in Linux distributions, and (2) our constructions rely on options that are GNU extensions. While the options we use for `find` + `mkdir` are also found in BSD `find` (macOS/FreeBSD), subtle runtime differences prevent our constructions from working as-is, though they may work with some modifications.

Table 1 summarizes the Turing completeness results established in this paper.

Table 1 Turing completeness results for `find`.

`find` Implementation	Key features	Model	Turing complete	Notes
GNU ($\geq$ 4.9.0)	`-files0-from`	`find` only	**Yes**	Sec. 4
GNU ($\geq$ 4.2.12)	`-regex -execdir`	`find` + `mkdir`	**Yes**	Sec. 3
GNU ($\geq$ 4.2.12)	`-regex -execdir`	`find` + `mkdir`	**Yes**[†]	Sec. 5
POSIX specification	-	`find` + `mkdir`	**Unknown**	No `-regex`

[†] Proven even without using regex back-references.

The question of whether similar computational universality exists in other implementations, such as BSD, BusyBox, and uutils `find`, remains an intriguing avenue for future research.

2 Preliminaries

To establish that computation arises solely from `find` and `mkdir`, we define the following execution model.

▶ **Definition 1.** *Let $\mathcal{B} = \{$`find`, `mkdir`$\}$ (for the `find` + `mkdir` system) or $\mathcal{B} = \{$`find`$\}$ (for the `find` system) be the set of permissible binaries. Fix a fresh, writable working directory "." that is initially empty.*

An execution *is a finite sequence of command invocations $C_0, \cdots, C_{m-1}$, where each C_i is an argument vector (program name and its arguments) that is fully determined before the execution begins – no command's arguments depend on the output of a previous command. The execution is valid if and only if every program executed during the entire run (including programs spawned indirectly via `-exec` or `-execdir` actions) is a binary in $\mathcal{B}$. The* result *of the execution is the concatenation of the standard output of the commands in execution order. We say the execution* halts *if all commands in the sequence terminate.*

This condition can be intuitively understood through a security lens: consider a restricted, shell-free system (e.g., a minimal container) that provides *only* the binaries in $\mathcal{B}$. An adversary may invoke a constant number of these commands. Although `-exec` and `-execdir` can in general launch arbitrary programs, in our model only $\mathcal{B}$ are available, so the adversary cannot exploit these actions to run anything beyond `find` and `mkdir`. What we prove is that even under this restriction, the adversary can force the system to perform arbitrary computations.

We clarify the resource model used throughout this paper.

▶ **Definition 2** (Resource model)**.** *We work in an idealized file-system model with unbounded capacity: there is no fixed upper bound on the number of directory entries that can be created, no bound on the total path length, and no bound on the amount of data stored in file contents. We do not, however, assume unbounded length of a single path component (i.e., a file or directory name): our constructions use only $O(1)$-length path components and therefore do not rely on increasing `NAME_MAX` (the operating system's limit on path component length, typically 255 bytes on Linux), which we treat as a fixed finite constant.*

Before presenting our constructions, we review the key concepts of GNU `find` [6] and `mkdir`.

The `find` utility recursively traverses a directory tree, evaluating an *expression* for each file visited. This expression functions as a domain-specific language for file filtering and manipulation. The expression is composed of:

- **Tests**: Predicates that return true or false based on file attributes (e.g., `-name`, `-empty`, `-regex`).
- **Actions**: Operations that perform side effects (e.g., `-exec`, `-delete`) or control traversal (e.g., `-prune`, `-quit`).
- **Operators**: Logical connectives `-and` (implicit), `-or` (`-o`), and `-not` (`!`), where and/or evaluations are short-circuiting (right-hand side is evaluated only if needed).

Of particular importance are the actions `-exec` and `-execdir`. They accept a command template terminated by `;`, where the string `{}` is replaced by the current file path. The specified (either `find` or `mkdir`) command is executed directly (via fork and exec) without invoking a shell, adhering to our execution model. The action is evaluated to true only if the exit status of the command is 0. `-exec` runs the command in the same directory as the outer `find`, and `-execdir` runs the command in the directory holding the current file, with `{}` replaced with the filepath relative from the directory. This locality of `-execdir` is vital for our scale-independent loop construction (Section 3.1), as it allows us to avoid the `PATH_MAX` limit.

We note that nesting `-exec` within `-exec` (e.g., `-exec find ... -exec ... ; ;`) is syntactically invalid: the outer `find` treats the first `;` as the terminator for its own `-exec`, so the inner `-exec` has no terminator. The same applies to `-execdir`. This restriction significantly constrains our design.

The directory traversal order of `find` is pre-order by default, meaning a directory is visited before its children. This can be changed to post-order with the global option `-depth`. Table 2 summarizes the `find` options used in this paper.

`mkdir` *path* creates a directory at the path. If its parent directory does not exist, creation fails and the command exits with a non-zero status. `mkdir -p` *path* creates the directory and its parent directories if they do not exist, and exits with a zero status. These functionalities are specified in POSIX and consistent across platforms. Our construction never relies on creation failure due to permission issues.

Table 2 Summary of GNU `find` options used in this paper. Operators below `(` *expr* `)` are listed in order of decreasing precedence.

Option/Operator	Description	Added in
`-true`	True.	-
`-empty`	True if the file/directory is empty.	-
`-name` *pattern*	True if the file's basename matches the pattern.	-
`-size` *n*`c`	True if the file size is *n* bytes.	-
`-regex` *pattern*	True if the file path matches the pattern.	-
`-exec` *cmd* `;`	Executes *cmd* with `{}` replaced by the path. True if the exit status is 0.	-
`-execdir` *cmd* `;`	Executes *cmd* in the file's directory with `{}` replaced by the relative path. True if the exit status is 0.	4.2.12
`-prune`	True. Do not descend into the matched directory.	-
`-quit`	Terminates the entire `find` execution.	-
`-delete`	Removes the matched file or (if empty) directory. Implies `-depth`. True if removal succeeds.	4.2.3
`-printf` *fmt*	True. Print formatted output to standard output.	-
`-fprintf` *file fmt*	True. Write formatted output to *file*.	-
`-depth`	Global option to do post-order traversal.	-
`-files0-from` *file*	Global option to read starting points from *file* (NUL-separated) instead of the command line.	4.9.0
`(` *expr* `)`	True if *expr* is true. Force precedence.	-
`!` *expr*	Not (true if *expr* is false).	-
expr1 expr2	And. *expr2* is not evaluated if *expr1* is false.	-
expr1 `-o` *expr2*	Or. *expr2* is not evaluated if *expr1* is true.	-

3 Turing Completeness of `find` + `mkdir`

In this section, we show the following theorem.

▶ **Theorem 3.** *GNU* `find` + `mkdir` *is Turing complete for* `find` *version* $\geq$ *4.2.12 (as of this writing when 4.10.0 is the latest), assuming the resource model in Definition 2.*

3.1 Loop Construction

The foundation of our constructions is the ability to create an infinite loop using only `find` and `mkdir`. To achieve this, we rely on a specific property of `find`: child directories created during the traversal of a directory become visible and are subsequently visited in the same execution, provided the traversal is pre-order (the default). When `find` visits a directory D

and an action creates a new subdirectory C inside D, C is visible to the subsequent directory enumeration and will be visited in the same `find` execution. While POSIX leaves this behavior unspecified[2], this behavior is consistent across GNU `find` versions[3].

The following code implements an infinite loop. In our code snippets, we do not escape characters like `;` (e.g., as `\;`), which is typically required by a shell. This is to signify shell independence. Note that proper escaping is necessary to execute these snippets in a shell.

```
mkdir x
find x -execdir mkdir {}/x ;
```

This code creates `x`, then `find` visits `x` and executes `mkdir` command in the directory `.` (the parent of `x`) with the argument `./x/x`, because `{}` is substituted with `./x`, that is the relative path of `x` from `.` . This creates `x/x`. Due to the aforementioned property, `find` then visits `x/x`, and it then creates `x/x/x`, and so on indefinitely.

We use `-execdir`, which was added in GNU `find` 4.2.12. While strictly speaking our resource model (Definition 2) does not forbid arbitrary long arguments, using `-execdir` is practically crucial. If we use `-exec`, `find` passes the path of the current file to the command. As the directory structure grows deeper, this path eventually exceeds `PATH_MAX` (typically 4096 bytes), causing `mkdir` to fail and `find` to halt. `-execdir` avoids this by executing the command in the subdirectory containing the current file, allowing us to pass an argument of constant length.

3.2 Tag System Implementation

We prove that `find` + `mkdir` is Turing complete by simulating a 2-tag system [12]. The following is a definition of a 2-tag system equivalent to the one given by Rogozhin [13].

▶ **Definition 4.** *A* 2-tag system *is defined by a pair* (Σ, P)*, where* Σ *is a finite alphabet containing a* halting symbol $H \in \Sigma$*, and* P *is a (partial) map with domain* $\Sigma \setminus \{H\}$ *and codomain* Σ^**. A* word *is a finite string over* Σ*. The computation starts with an* initial word $w_1 \in \Sigma^*$*, and proceeds iteratively. In the* i*-th iteration, with* $w_i = x_{i,1} \cdots x_{i,k}$*, the computation halts if* $k < 2$ *or* $x_{i,1} = H$*. Otherwise, the next word is* $w_{i+1} = x_{i,3} \cdots x_{i,k}\, P(x_{i,1})$*. If the computation halts at* w_t*,* w_t *is called the* halting word*, and is the* output *of the computation.*

It is known that 2-tag systems are computationally universal. Specifically, we can simulate an arbitrary 2-symbol m-state Turing machine ($\mathrm{TM}(m, 2)$) using a 2-tag system. Combining this reduction with the known existence of a universal Turing machine with 18 states and 2 symbols ($\mathrm{TM}(18, 2)$) [10], De Mol [9] constructed a universal 2-tag system with 576 symbols ($\mathrm{TS}(576, 2)$).

We prove Turing completeness of `find` + `mkdir` by simulating an arbitrary 2-tag system, applying the construction to this 576-symbol universal instance. The high-level idea is:

1. Encode each tag-system symbol as a two-letter directory name (e.g., `/ab`), representing the sequence of words as a growing, nested directory path.
2. To compute the next word, iteratively copy the remaining suffix of the previous word symbol-by-symbol: use a regex with a back-reference to verify the progress and locate the next symbol, then use `-execdir mkdir` to append it to the path.
3. Once the suffix is fully copied, use regex matching to identify the first symbol of the previous word, and append its corresponding production string to complete the iteration.

[2] "If a file is removed from or added to the directory hierarchy being searched it is unspecified whether or not find includes that file in its search." [14]
[3] Verified on GNU find 4.10.0 (Debian) and 4.5.11 (CentOS 7).

We will encode the input word to an ASCII string and decode the halting word from an ASCII string. Since this mapping is straightforward (clearly computable by a deterministic finite state transducer), we omit a formal proof for the validity of the encoding. We use `typewriter` font to denote string literals, concatenation (written by placing strings next to each other), and ε for the empty string.

Let $\mu = 576$, and $\Sigma = \{s_1, \ldots, s_\mu, s_{\mu+1} = H\}$. We encode each symbol as a directory name: let A be a set of lowercase ASCII letters, and choose distinct strings $s'_1, \ldots, s'_{\mu+1} \in A^2$ (possible because $577 < 26^2 = 676$). Define $\sigma_k = \texttt{/}\,s'_k$, where `/` is the directory separator. The encoding of a symbol s_k is thus a three-character string like `/ab`. We let $\Sigma' = \{\sigma_1, \ldots, \sigma_\mu, \sigma_{\mu+1} = \eta\}$ and $\phi : \Sigma \to \Sigma'$ be a function that maps s_i to σ_i. We also define $\Phi : \Sigma^* \to \Sigma'^*$ so that $\Phi(\varepsilon) = \varepsilon$ and $\Phi(s_1 \cdots s_k) = \Phi(s_1 \cdots s_{k-1})\,\phi(s_k)$ for $k \geq 1$. We define $\pi(s_i) = \Phi(P(s_i))$ for $i \leq \mu$.

We first run the following command to embed the initial word w_1. We use spaces to separate command arguments, distinguishing them from concatenation.

```
mkdir -p _Φ(w1)/_
```

The idea for the following computation is to iteratively append the file path representing the next word to it using `/_` as a separator. During the computation, the working directory should contain exactly one directory `_`, each of whose descendants is a directory containing at most one directory. We call the path to the empty directory at a point of the computation the state of the system at that point. After the computation halts, the state of the system should be $\texttt{\_}\,\Phi(w_1)\,\texttt{/\_}\,\Phi(w_2)\,\texttt{/\_} \cdots \texttt{/\_}\Phi(w_t)\,\texttt{/\_}$, and during the computation the state should be a prefix of it. Let $\Theta_{i,j}$ be the following state. Our initial state is $\Theta_{1,0}$ and the last state will be $\Theta_{t,0}$.

$$\Theta_{i,j} = \texttt{\_} \cdots \texttt{\_}\,\Phi(w_i)\,\texttt{/\_}\,\phi(x_{i+1,1}) \cdots \phi(x_{i+1,j})$$

At $\Theta_{i,j}$, if $j = 0$ and w_i is a halting word, we stop the computation. Otherwise, (1) if $\phi(x_{i+1,1}) \cdots \phi(x_{i+1,j})$ equals $\phi(x_{i,3}) \cdots \phi(x_{i,j+2})$ and $x_{i,j+3}$ exists, we append $\phi(x_{i,j+3})$ to the state, and (2) otherwise (if we have copied the last alphabet of the previous word), we append $\pi(x_{i,1})\texttt{/\_}$ to the state.

Let the regex pattern $\lambda = \texttt{/[a-z][a-z]}$. Since `[a-z]` matches any lowercase ASCII letter but not `/` or `_`, λ matches exactly the strings in Σ' (encoded symbols). Let $\Lambda = \texttt{\textbackslash(}\,\lambda\,\texttt{\textbackslash)*}$ be a regex that matches any (possibly empty) sequence of encoded symbols. `\(` and `\)` are escaped because a backslash before parentheses is required by `find`'s default regex type (emacs regex). The same applies for `\|` (OR operator) below. Define the following regex patterns:

$$\alpha_k = \texttt{.*\_}\,\lambda\,\lambda\,\texttt{\textbackslash(}\,\Lambda\,\texttt{\textbackslash)}\,\sigma_k\,\Lambda\,\texttt{/\_\textbackslash1}$$
$$\beta_k = \texttt{.*\_}\,\sigma_k\,\Lambda\,\texttt{/\_}\,\Lambda$$
$$\gamma = \texttt{.*\_\textbackslash(\textbackslash|}\,\lambda\,\texttt{\textbackslash|}\,\eta\,\Lambda\,\texttt{\textbackslash)/\_}$$

We run the following as the second command. Newlines are for readability and have the same meaning as spaces.

```
find _ -empty (
   -regex γ -quit -o
   -regex α1 -execdir mkdir {}σ1 ; -o
   ...
   -regex αμ+1 -execdir mkdir {}σμ+1 ; -o
```

```
    -regex β1 -execdir mkdir -p {}π(s1)/_ ; -o
    ...
    -regex βμ -execdir mkdir -p {}π(sμ)/_ ; -o
    -printf unreachable
)
```

Finally we run the following command to output the encoded result.

```
find _ -depth ! -empty -name _ -execdir find _ ! -name _ -printf /%f ; -quit
```

We show that the output of the last command is $\Phi(w_t)$ if the original 2-tag system halts with w_t, and the second command does not halt if the original does not halt, proving Theorem 3.

Proof. Let $\mathcal{L}(r)$ be the language denoted by the regex r. Remember that $\Sigma' \subset \mathcal{L}(\lambda), \Sigma'^* \subset \mathcal{L}(\Lambda)$ and any string including `_` or ending with `/` is not in $\mathcal{L}(\lambda)$ nor $\mathcal{L}(\Lambda)$. $\{\sigma_k\} = \mathcal{L}(\sigma_k)$ and $\{\eta\} = \mathcal{L}(\eta)$. We have $\sigma_k = \phi(s_k)$ and $\Theta_{i,j} = \cdots \_\, \phi(x_{i,1}) \cdots \phi(x_{i,|w_i|})/\_\, \phi(x_{i+1,1}) \cdots \phi(x_{i+1,j})$, where $x_{i+1,j+2} = x_{i,j}$ for $1 \le j \le |w_i| - 2$.

We prove that the state is $\Theta_{t,0}$ if the second command halts by induction. Let us call the expression in the parentheses the *main expression*. As the base case, `find` visits $\Theta_{1,0}$ and evaluates the main expression for the first time because of the `-empty` predicate. Assume the current state is $\Theta_{i,j}$ and `find` visits it, evaluating the main expression. By construction, γ matches if and only if $j = 0$ and $|w_i| < 2$ or $x_{i,1} = H$, meaning w_i is the halting word. If $j < |w_i| - 2$ and $x_{i,j+3} = s_k$, $\Theta_{i,j} \in \mathcal{L}(\alpha_k)$ because σ_k matches with $\phi(s_k)$, and `\(` Λ `\)` and `\1` match with $\phi(x_{i,3}) \cdots \phi(x_{i,j+2})$ and $\phi(x_{i+1,1}) \cdots \phi(x_{i+1,j})$ which are the same string. The converse can be said because the `_` before `\1` combined with the fact the same string should match Λ constrains it to match $\phi(x_{i+1,1}) \cdots \phi(x_{i+1,j})$ and σ_k only matches with $\phi(s_k)$. Provided $j = |w_i| - 2$, $\Theta_{i,j} \in \mathcal{L}(\beta_k)$ if and only if $\phi(x_{i,1}) = \sigma_k$ by construction. These enforce the main expression to work as follows: (1) If $(i, j) = (t, 0)$, it halts the command. (2) Otherwise if $j < |w_i| - 2$, it executes `mkdir {}` $\phi(x_{i,j+3})$ turning the state into $\Phi_{i,j+1}$. (3) Otherwise at $j = |w_i| - 2$, it executes `mkdir -p {}` $\pi(x_{i,1})$`/_` turning the state into $\Phi_{i+1,0}$. The newly created state is subsequently visited due to the said GNU `find` property, so the induction completes.

The third command visits the state in post-order (`-depth`), matches the first non-empty `_`, which is the second-to-last `_` directory (say *base*), executes the inner `find` for the base directory, which visits its descendants in pre-order, printing the name of each directory prefixing `/` (due to `-printf /%f`) unless the name is `_` (due to `! -name _`), effectively outputting $\Phi(w_t)$, then quit. ◀

4 Turing Completeness of GNU `find` (Standalone)

We prove that GNU `find` version 4.9.0 or later is Turing complete *without* `mkdir`. We prove this by simulating Minsky's program machine [8].

▶ **Theorem 5.** *GNU `find` is Turing complete for `find` version ≥ 4.9.0 (as of this writing when 4.10.0 is the latest), assuming the resource model in Definition 2.*

The high-level idea is as follows. We represent two counters as files whose byte-length encodes their values. The key mechanism is the `-files0-from` option, which lets `find` read starting points from a file: if the file contains n copies of the string `.` NUL, `find` visits the current directory `.` exactly n times, allowing us to repeat an action n times. By reading and writing such files during traversal, we simulate the increment, decrement, and conditional-jump instructions of a two-counter program machine.

4.1 Loop Construction

The foundation of our constructions is the ability to create an infinite loop using only `find`. We utilize `-files0-from` *`file`*, which lets `find` read NUL-separated starting points from a file rather than the command line (where NUL is the ASCII null character). `find`'s `-fprintf` can write NUL to a file provided `\0` verbatim. By updating this file while the `find` command is running, we can supply an infinite stream of starting points, thereby creating a loop.

We first introduce a way of encoding a non-negative integer as a file. Let $\iota =$ `.` NUL (a two-byte string). If the content of a file f is the string ι repeated n times (i.e., $2n$ bytes), we say the *value* of f is n. For example, a file of value 3 contains the 6-byte string `.`NUL `.`NUL `.`NUL. If f does not exist, its value is 0; in other cases the value is undefined. We will use files `a`, `b`, `s`, `t` and represent their values with a, b, s, t respectively. We use t for temporary storage. We construct our program so that throughout the computation a, b, s are always defined.

Notice that the code like `find -files0-from s -prune` *`expr`* evaluates *expr* s times, and it exits with non-zero exit code if $s = 0$ (`s` does not exist).

For loop construction, we first initialize s (for stream) with 1.

```
1 find -fprintf s .\0 -quit
```

We define an expression which increments s when evaluated as follows, and call this INC(`s`). We similarly define INC(`a`) and INC(`b`) for later use, in which `s` is replaced with `a` or `b`.

```
1 ( ( -exec find s -quit ;
2 -exec find -quit -fprintf first . ;
3 -exec find -files0-from s ( -name . -o -name first -delete ) -fprintf t .\0 ;
4 -exec find -files0-from t -prune -fprintf s .\0 ;
5 -exec find t -delete ; ) -o
6 -exec find -fprintf s .\0 -quit ; )
```

To understand this, remember that the file given to `-fprintf` is truncated or created to 0 bytes as soon as the `find` command starts, regardless of whether its predicate ever matches (as documented). Line 1 is evaluated to true if and only if $s \neq 0$. Line 2 (where $s > 0$) creates an empty file `first` (because of `-quit`, `-fprintf` is not evaluated). Line 3 updates `t` with `.` NUL repeated $s + 1$ times and removes `first` (first evaluation matches with both `.` and `./first`, removing `first` with `-delete`), effectively meaning $t \leftarrow s + 1$. Line 4 updates `s` with `.` NUL repeated t times ($s \leftarrow t$), and line 5 deletes `t` ($t \leftarrow 0$). If $s = 0$, line 6 updates `s` with `.` NUL ($s \leftarrow 1$).

The loop is constructed as follows. Remember we start with $s = 1$.

```
1 find -files0-from s -prune INC(s);
```

This works because GNU `find` reads `s` as a stream. When the first `.` NUL is read, `find` still has not read the EOF (end of file). It then executes INC(`s`) to extend the file so that it still has the next `.` NUL, and this process continues indefinitely. The fact that this works depends entirely on the implementation of GNU `find`, but this has been empirically confirmed on GNU `find` 4.10.0 and rigorously confirmed to work by reading the source code of GNU `find` 4.9.0.

4.2 Program Machines

The following definition for 2-counter program machine is equivalent to the one given by Minsky [8]. 2-counter program machines are known to be Turing complete.

▶ **Definition 6.** *A 2-counter program machine is a finite sequence of instructions* $P = (I_0, \ldots, I_{m-1})$ *acting on counters* $c_0, c_1 \in \mathbb{N}$. *A configuration is* $(\mathrm{pc}, c_0, c_1) \in \{0, \ldots, m\} \times \mathbb{N}^2$, *where* $\mathrm{pc} = m$ *denotes the* halted *configuration. The one-step transition relation* $\to_P$ *is defined for* $\mathrm{pc} < m$ *as follows (with* $r \in \{0, 1\}$ *and* $q \in \{0, \ldots, m\}$*):*

- *If* $I_{\mathrm{pc}} = \mathrm{INC}(r)$ *then* $c_r \leftarrow c_r + 1$ *and* $\mathrm{pc} \leftarrow \mathrm{pc} + 1$.
- *If* $I_{\mathrm{pc}} = \mathrm{DEC}(r)$ *then* $c_r \leftarrow \max\{c_r - 1, 0\}$ *and* $\mathrm{pc} \leftarrow \mathrm{pc} + 1$.
- *If* $I_{\mathrm{pc}} = \mathrm{JZ}(r, q)$ *then* $\mathrm{pc} \leftarrow q$ *if* $c_r = 0$, *and* $\mathrm{pc} \leftarrow \mathrm{pc} + 1$ *otherwise.*
- *If* $I_{\mathrm{pc}} = \mathrm{J}(q)$ *then* $\mathrm{pc} \leftarrow q$.

An execution of P *from an initial configuration* $(0, c_0, c_1)$ *is the maximal* $\to_P$ *chain; it* halts *if and only if it reaches* $\mathrm{pc} = m$, *and outputs* c_0.

To simulate 2-counter program with `find`, we create expressions to simulate the instructions. We have already seen INC expressions. We define the following expression as DEC(`a`), and DEC(`b`) analogously.

```
1 ( ( -exec find a -size 2c -delete ;
2 -exec find a -quit ;
3 -exec find -quit -fprint first ;
4 -exec find -files0-from a -name first -delete -fprintf t skip -o -name t -
      fprintf t . -o -name . -fprintf t \0 ;
5 -exec find -files0-from t -name . -fprintf a .\0 ;
6 -exec find t -delete ;
7 ) -o -true )
```

When the expression is evaluated, it sets a to $\max(a - 1, 0)$ as follows: Line 1 evaluates to true and deletes `a` ($a \leftarrow 0$) if and only if $a = 1$ (since the file size is 2 bytes, checked via `-size 2c`). Line 2 evaluates to false if $a = 0$. Line 3 (with $a \geq 2$) creates `first`. Line 4 sets t to $a - 1$, which happens as follows: (1) Due to `-fprintf t`, `t` exists as an empty file as soon as the `find` starts. (2) In the first iteration of `.` from `a`, because of `-delete` implying `-depth`, `./first` (being deleted) and `./t` are iterated before `.` and it causes either `.skip` NUL or `skip.` NUL to be written to `t`. (3) In the following $a - 1$ iterations, since `first` no longer exists, $a - 1$ repeat of `.` NUL is appended to `t`. In line 5, since neither file `.skip` nor `skip.` exists, it is skipped with an error message written to standard error (which we do not use), causing `-fprintf a .\0` to be evaluated $a - 1$ times ($a \leftarrow a - 1$). Line 6 deletes `t` ($t \leftarrow 0$). Line 7 exists to make DEC(`a`) always true.

The program counter is represented by the file `pc`, but unlike a and b, we directly encode the value n as a string of n ones (ASCII character `1`). For example, the program counter being 3 means `pc`'s content is `111`. `pc` will be initialized to an empty file with `find -quit -fprintf pc not-written`, and the program counter will always be defined hereafter. Let $\mathtt{1}^n$ denote a string of n ones.

We define the expression J(q) for $q \in \{0, \ldots, m\}$ as follows. If an empty string is a concern, we can make it `-exec find -quit -fprintf pc x ;` for $q = 0$. It is clear that it is evaluated to true and sets pc to q.

```
1 -exec find -fprintf pc 1^q -quit ;
```

We define the expression JZ(`a`, q) for $q \in \{0, \ldots, m\}$ as follows, and JZ(`b`, q) analogously. It is clear that JZ(`a`, q) is evaluated to true and sets pc to q if and only if $a = 0$.

```
( ( ! -exec find a -quit ; J(q) ) -o -true )
```

Finally, we define the expression ISPC(q) for $q \in \{0, \ldots, m-1\}$ as follows. Unlike other expressions, it assumes the current file is `pc`, and is evaluated to true if and only if $\mathrm{pc} = q$. q in the code is substituted to the corresponding decimal number represented in ASCII.

```
-size qc
```

The commands to simulate a given 2-counter program $P = (I_0, \ldots, I_{m-1})$ and initial configuration $(0, c_0, c_1)$ is as follows. We let $\gamma_0 =$ `a` and $\gamma_1 =$ `b`.

Our first command to run is `find -quit -fprintf pc x`, which initializes pc to 0. The next command to run is the following, where INC(`a`) is repeated c_0 times and INC(`b`) is repeated c_1 times. This initializes s, a, and b with 1, c_0, and c_1 respectively.

```
find INC(s) INC(a) ··· INC(a) INC(b) ··· INC(b) -quit
```

The next command is the main loop. Let ϕ be a mapping from $\{I_0, \ldots, I_{m-1}\}$ to `find` expressions defined by the following.

$$\begin{aligned}
\phi(\mathrm{INC}(r)) &= \mathrm{INC}(\gamma_r) \\
\phi(\mathrm{DEC}(r)) &= \mathrm{DEC}(\gamma_r) \\
\phi(\mathrm{JZ}(r, q)) &= \mathrm{JZ}(\gamma_r, q) \\
\phi(\mathrm{J}(q)) &= \mathrm{J}(q)
\end{aligned}$$

The main loop is as follows.

```
find -files0-from s -name pc INC(s) (
   ISPC(0) J(1) φ(I_0) -o
   ···
   ISPC(m − 1) J(m) φ(I_{m−1}) -o
   -quit
)
```

Finally, the following two commands allow us to directly output the value of a. The first command writes a `1`s to `count`. The second command outputs the byte count of `count`, which is a.

```
find -files0-from a -fprintf count 1 -prune
find count -printf %s
```

We show that by running the above commands from the first to the last, we get the same output as the original 2-counter machine if the machine halts, and it does not halt if the machine does not halt, completing the proof of Theorem 5. It is enough to show that the main loop works as expected.

Proof for Theorem 5. We call the expression in the parenthesis the *main expression*. The main loop uses `-files0-from s` to read starting points from the file `s`: each `.` NUL entry in `s` causes `find` to visit the current directory `.`, where it finds the file `pc` and evaluates INC(`s`) followed by the main expression. Since INC(`s`) appends a new entry to `s` before `find` reaches the end of the file, the loop runs indefinitely (until `-quit` is evaluated).

We prove the claim by induction on the number of steps t of the program machine. Let $(\mathrm{pc}^{(t)}, c_0^{(t)}, c_1^{(t)})$ be the configuration of the program machine at step t. Assume that the values encoded by `pc`, `a`, `b` (pc, a, b) are equal to $\mathrm{pc}^{(t)}, c_0^{(t)}, c_1^{(t)}$ just before the $(t+1)$-th evaluation of the main expression. The base case $t = 0$ holds as previously shown.

At the $(t+1)$-th evaluation, $\textsc{ispc}(i)$ is true if and only if $i = \mathrm{pc} = \mathrm{pc}^{(t)}$. Therefore, `-quit` is evaluated if and only if $\mathrm{pc} = m$, in which case both the main loop and the program machine halt, leaving $a = c_0^{(t)}$. Otherwise, the program machine executes I_{pc}, and the `find` command evaluates $\textsc{j}(\mathrm{pc}+1)$ $\phi(I_{\mathrm{pc}})$. This sequence updates (pc, a, b) exactly as I_{pc} updates $(\mathrm{pc}^{(t)}, c_0^{(t)}, c_1^{(t)})$, so the new values equal $(\mathrm{pc}^{(t+1)}, c_0^{(t+1)}, c_1^{(t+1)})$. Since $\textsc{j}(\mathrm{pc}+1)$ $\phi(I_{\mathrm{pc}})$ always evaluates to true, the main expression completes (due to the subsequent `-o`). This completes the proof. ◀

5 Turing Completeness of `find` + `mkdir` without Back-references (Sketch)

While Section 3 relies on regex back-references for the "copy" operation, we can achieve Turing completeness *without* back-references. We present the core ideas for our proof and provide a proof sketch. The full proof is in Appendix A.

To signify non-use of back-references, we will always use `-regextype awk` global option for any `find` that uses `-regex` ***pattern***. The awk regex engine does not support back-references. (It also uses unescaped parentheses `(` and `)` for grouping, unlike the default emacs regex engine which requires `\(` and `\)`, though this is not an important advantage for us, and the similar construction should work for any other regex types.)

We build on the idea in Section 3 to simulate 2-tag systems. That is, during the computation, there is only one empty directory under the current working directory, and the path to it represents the state of the computation. The encoded symbols are the strings $\{\sigma_i\}$. When the computation halts, the final state encodes the sequence of words $w_1, \ldots, w_t$ from left to right, delimited by separator strings. During the computation, the state has the form of $\Theta_{i,j}$, which is a prefix of the last state and encodes the status where the word w_i is given and the first j letters of the word w_{i+1} have been computed. $x_{i,j}$ denotes the j-th symbol of w_i, and we let ϕ be a function that maps s_i to σ_i, where $\{s_i\}$ are the symbols of the original tag-system.

We define $\Sigma' = \{\sigma_1, \ldots, \sigma_{\mu+1}\}$ as before. That is, $\Sigma' \subset \mathcal{L}(\texttt{/[a-z][a-z]})$ and $\sigma_i \neq \sigma_j$ for $i \neq j$. The main trick is that we use $\textsc{sep} = \texttt{\$\_)?(}$ as the separator, instead of `_`. Those are allowed characters for filenames. Note that the letter `$` in regex is special in that it only matches the end of the string, and `?` indicates one or zero occurrence of the previous atom. This means `$_` is a pattern that never matches any string. This choice makes the state $\Theta_{i,j}$ during the computation look like the following.

$$\Theta_{i,j} = \texttt{\$\_)?(}\cdots\texttt{/\$\_)?(}\,\phi(x_{i,1})\cdots\phi(x_{i,|w_i|})\,\texttt{/\$\_)?(}\,\phi(x_{i+1,1})\cdots\phi(x_{i+1,j})$$

Let us consider the regex `(`$\Theta_{i,j}$`)`, which should look like this:

$$\texttt{(\$\_)?(}\cdots\texttt{\$\_)?}\cdots\texttt{(}\cdots\texttt{\$\_)?(}\,\phi(x_{i+1,1})\cdots\phi(x_{i+1,j})\,\texttt{)}$$

This (valid) regex matches only $\phi(x_{i+1,1})\cdots\phi(x_{i+1,j})$. The regex obviously matches the string, and conversely, if it matches (because $\mathcal{L}(\cdots\texttt{\$\_}) = \emptyset$), every `(`$\cdots$`)?` must be matching with an empty string.

This observation allows us to update the part where we used back-references in the previous proof. That is, at state $\Theta_{i,j}$ where $j < |w_i| - 2$, we append $\phi(x_{i,j+3})$ to the state without back-references. The idea is (1) we create $\mu+1$ marker files $t_1, \ldots, t_{\mu+1}$ under the directory that represents the current state, (2) we invoke $\mu+1$ inner `find`'s so that k-th one's regex matches with t_k's path only when $\phi(x_{i,j+3}) = \sigma_k$, deleting the marker file. To construct such a regex, we embed the state into the regex with `{}` (detailed below). (3) we check marker files and create a corresponding directory if a file is gone, advancing the state.

The regex constructed in the step (2) for k is the following, where $\lambda = \texttt{/[a-z][a-z]}$ and $\Lambda = \texttt{(}\lambda\texttt{)*}$.

$$\texttt{.*}\,\text{SEP}\,\lambda\,\lambda\,\texttt{(\{\})}\,\sigma_k\,\Lambda\,\text{SEP}\Lambda\,\texttt{/}\,t_k$$

The state $\Theta_{i,j}$ has $\phi(x_{i+1,1}) \cdots \phi(x_{i+1,j})$ after the separator. The `({})` is expanded to `(`$\Theta_{i,j}$`)`, and as we have seen it matches with and only with $\phi(x_{i+1,1}) \cdots \phi(x_{i+1,j})$. We know it is equal to $\phi(x_{i,3}) \cdots \phi(x_{i,j+2})$. This ensures that the regex matches if and only if $\phi(x_{i,j+3}) = \sigma_k$, and the current file is the marker file t_k.

We can use the same construction as the previous proof on other parts of the commands, and this completes the proof sketch of the following theorem.

▶ **Theorem 7.** *GNU* ***find*** *+* ***mkdir*** *is Turing complete without using regex back-references for* ***find*** *version ≥ 4.2.12 (as of this writing when 4.10.0 is the latest), assuming the resource model in Definition 2.*

References

1 Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. The page-fault weird machine: Lessons in instruction-less computation. In Jon Oberheide and William K. Robertson, editors, *7th USENIX Workshop on Offensive Technologies, WOOT '13, Washington, D.C., USA, August 13, 2013*. USENIX Association, 2013. URL: https://www.usenix.org/conference/woot13/workshop-program/presentation/bangert.

2 Christophe Blaess. turing.sed. https://sed.sourceforge.net/grabbag/scripts/turing.sed, 2001. Link dead. Archived from the original on 2018-01-16: https://web.archive.org/web/20180116201401/http://sed.sourceforge.net/grabbag/scripts/turing.sed. Accessed: 2026-01-10.

3 Christophe Blaess. Implementation of a Turing Machine as Sed script. https://sed.sourceforge.net/grabbag/scripts/turing.txt, 2003. Link dead. Archived from the original on 2018-02-20: https://web.archive.org/web/20180220011912/http://sed.sourceforge.net/grabbag/scripts/turing.txt. Accessed: 2026-01-10.

4 Alex Churchill, Stella Biderman, and Austin Herrick. Magic: The gathering is turing complete. In Martin Farach-Colton, Giuseppe Prencipe, and Ryuhei Uehara, editors, *10th International Conference on Fun with Algorithms, FUN 2021, Favignana Island, Sicily, Italy, May 30 - June 1, 2021*, volume 157 of *LIPIcs*, pages 9:1–9:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.FUN.2021.9.

5 Stephen Dolan. mov is Turing-complete. https://stedolan.net/research/mov.pdf, 2013. Link dead. Archived from the original on 2021-02-14: https://web.archive.org/web/20210214020524/https://stedolan.net/research/mov.pdf. Accessed: 2026-01-03.

6 Free Software Foundation. *GNU Findutils Manual*, 2024. Version 4.10.0. URL: https://www.gnu.org/software/findutils/manual/.

7 Hackaday. Proof that find + mkdir are Turing-Complete. https://hackaday.com/2024/08/05/proof-that-find-mkdir-are-turing-complete/, 2024. Accessed: 2026-01-10.

8 Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., 1967.

9 Liesbeth De Mol. Tag systems and collatz-like functions. *Theor. Comput. Sci.*, 390(1):92–101, 2008. doi:10.1016/j.tcs.2007.10.020.

10 Turlough Neary and Damien Woods. Four small universal turing machines. *Fundam. Informaticae*, 91(1):123–144, 2009. doi:10.3233/FI-2009-0036.

11 Keigo Oka. find + mkdir is Turing complete. https://ogiekako.vercel.app/blog/find_mkdir_tc, 2024. Accessed: 2026-01-10.

12 Emil L. Post. Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65(2):197–215, 1943.

13 Yurii Rogozhin. Small universal turing machines. *Theor. Comput. Sci.*, 168(2):215–240, 1996. doi:10.1016/S0304-3975(96)00077-1.

14 The Open Group. *The Open Group Base Specifications Issue 8, 2024 Edition: find*, 2024. Accessed: 2026-01-10. URL: https://pubs.opengroup.org/onlinepubs/9799919799/utilities/find.html.

A Turing Completeness of `find` + `mkdir` without Back-references

We expand on the sketch of the proof in Section 5.

We prove Theorem 7 by simulating 2-tag systems (Definition 4) with `find` + `mkdir` without regex back-references. We use the notation in the definition in the following proof. Note that the construction resembles the one in Section 3.

Let $\mu = 576$, and $\Sigma = \{s_1, \dots, s_\mu, s_{\mu+1} = H\}$. Let A be a set of lowercase ASCII letters, and $\{s'_1, \dots, s'_\mu, s'_{\mu+1}\} \subset A^2$ be distinct strings with length 2. We will use `-regextype awk` for `find`s with `-regex` to ensure that regexes are matched without back-references, while similar constructions should work for other regex types.

We define $\sigma_k = \texttt{/}\, s'_k$, where `/` is the ASCII letter for directory separator. We let $\Sigma' = \{\sigma_1, \dots, \sigma_\mu, \sigma_{\mu+1} = \eta\}$ and $\phi : \Sigma \to \Sigma'$ be a function that maps s_i to σ_i. We also define $\Phi : \Sigma^* \to \Sigma'^*$ so that $\Phi(\varepsilon) = \varepsilon$ and $\Phi(s_1 \cdots s_k) = \Phi(s_1 \cdots s_{k-1})\, \phi(s_k)$ for $k \geq 1$. We define $\pi(s_i) = \Phi(P(s_i))$ for $i \leq \mu$.

Let $\mathrm{SEP} = \texttt{\$\_)?(}$. We first run the following command to embed the initial word.

```
mkdir -p SEP Φ(w_1) / SEP
```

The idea for the following computation is to iteratively append the file path representing the next word to it using SEP as a separator. During the computation, the current working directory (CWD) should contain exactly one directory SEP, and under SEP there is exactly one empty directory. We call the path to the empty directory at a point of the computation the state of the system at that point. After the computation halts, the state of the system should be $\mathrm{SEP}\, \Phi(w_1) / \mathrm{SEP}\, \Phi(w_2) / \mathrm{SEP} \cdots / \mathrm{SEP}\, \Phi(w_t) / \mathrm{SEP}$, and during the computation the state should be a prefix of it. Let $\Theta_{i,j}$ be the following state. Our initial state is $\Theta_{1,0}$ and the last state will be $\Theta_{t,0}$.

$$\Theta_{i,j} = \cdots \mathrm{SEP}\, \Phi(w_i) / \mathrm{SEP}\, \phi(x_{i+1,1}) \cdots \phi(x_{i+1,j})$$

At $\Theta_{i,j}$, if $j = 0$ and w_i is a word to halt, we stop the computation. Otherwise, (1) if $\phi(x_{i,j+3})$ exists, we append $\phi(x_{i,j+3})$ to the state, and (2) otherwise (if we have copied the last alphabet of the previous word), we append $\pi(x_{i,1}) / \mathrm{SEP}$ to the state.

Let $\lambda = \texttt{/[a-z][a-z]}$ be a regex that matches any string in Σ'. Let $\Lambda = \texttt{(}\lambda\texttt{)*}$ be a regex that matches any repetition of strings in Σ'. We also define $\overline{\mathrm{SEP}} = \texttt{\textbackslash\$\_\textbackslash)\textbackslash?\textbackslash(}$ as a string that satisfies $\mathcal{L}(\overline{\mathrm{SEP}}) = \{\mathrm{SEP}\}$. Define $t_1 = \texttt{1}, t_2 = \texttt{2}, \dots$ until $t_{\mu+1}$, which is an ASCII string representing $\mu + 1$ in decimal. Define the following strings.

$$\alpha_k = \texttt{.*}\, \overline{\mathrm{SEP}}\, \lambda\, \lambda\, \texttt{({})}\, \sigma_k\, \Lambda / \overline{\mathrm{SEP}}\, \Lambda / t_k$$
$$\beta_k = \texttt{.*}\, \overline{\mathrm{SEP}}\, \sigma_k\, \Lambda / \overline{\mathrm{SEP}}\, \Lambda$$
$$\gamma = \texttt{.*}\, \overline{\mathrm{SEP}}\, \texttt{(|}\, \lambda\, \texttt{|}\, \eta\, \Lambda\, \texttt{)} / \overline{\mathrm{SEP}}$$

We run the following as the second command. Newlines are for readability and have the same meaning as spaces.

```
1  find SEP -regextype awk -type d -empty (
2     -regex γ -quit -o                                          ▷ Halt check
3     -execdir find -fprint {}/t_1 -quit ;                       ▷ Save t_k
4     ...
5     -execdir find -fprint {}/t_{μ+1} -quit ;
6     -exec find SEP -regextype awk -type f -regex α_1 -delete -quit ;   ▷ Check α_k
7     ...
8     -exec find SEP -regextype awk -type f -regex α_{μ+1} -delete -quit ;
9     (
10       ! -execdir find {}/t_1 -quit ; -execdir mkdir {}/σ_1 ; -o      ▷ Append next
11       ...
12       ! -execdir find {}/t_{μ+1} -quit ; -execdir mkdir {}/σ_{μ+1} ; -o
13       -false
14    ) -o (
15       -regex β_1 -execdir mkdir -p {}π(s_1)/SEP ; -o                 ▷ Transition
16       ...
17       -regex β_μ -execdir mkdir -p {}π(s_μ)/SEP ; -o
18       -printf unreachable
19    ) ,
20    -exec find SEP -type f -delete ;
21 )
```

It uses some constructions that are not mentioned in the preliminaries section: `-type d` and `-type f` are evaluated to true if and only if the current file is a directory and a regular file, respectively. `-fprint` *`file`* prints the path to the current file to *file* (while we only use it to create a file). *`expr1`* `,` *`expr2`* evaluates both *expr1* and *expr2* and its value becomes the value of *expr2*. The `,` has the least precedence (less than `-o`).

Finally we run the following command to output the encoded result.

```
find SEP -depth ! -empty -name SEP -execdir find SEP ! -name SEP -printf /%f
    ; -quit
```

We show that the output of the last command is $\Phi(w_t)$ if the original 2-tag system halts with w_t, and the second command does not halt if the original does not halt, proving Theorem 7.

Proof. Remember that $\Sigma' \subset \mathcal{L}(\lambda), \Sigma'^* \subset \mathcal{L}(\Lambda)$ and any string including a character in SEP or ending with / is not in $\mathcal{L}(\lambda)$ nor $\mathcal{L}(\Lambda)$. $\{\sigma_k\} = \mathcal{L}(\sigma_k)$ and $\{\eta\} = \mathcal{L}(\eta)$. We have $\Theta_{i,j} = \cdots \text{SEP}\, \phi(x_{i,1}) \cdots \phi(x_{i,|w_i|})\, /\, \text{SEP}\, \phi(x_{i+1,1}) \cdots \phi(x_{i+1,j})$, where $x_{i+1,j+2} = x_{i,j}$ for $1 \leq j \leq |w_i| - 2$.

We prove that the state is $\Theta_{t,0}$ if the second command halts by induction. Let us call the expression in the outermost parentheses the *main expression*. The `-type d` predicate is evaluated to true if and only if the current file is a directory. This means the expression is evaluated only when the current file is an empty directory. As the base case, `find` visits $\Theta_{1,0}$ and evaluates the main expression for the first time. Assume the current state is $\Theta_{i,j}$ and `find` visits it, evaluating the main expression. We also assume that before the evaluation $\Theta_{i,j}$ is the only empty directory under CWD, and there is no regular file (that matches `-type f`) under CWD, calling it the *cleanness condition*. We will prove that after the evaluation the cleanness condition still holds as well. By construction, γ matches and halts computation if and only if $j = 0$ and $|w_i| < 2$ or $x_{i,1} = H$, meaning w_i is the halting word (Line 2). If it does not halt, it creates marker files $\Theta_{i,j}\, /\, t_k$ for $k = 1, \ldots, \mu + 1$ (Line 3). Then it evaluates the expressions from line 6. The k-th expression from line 6 executes `find` for the SEP in

the CWD. Due to the cleanness condition and `-type f`, if it matches, it should be one of $t_1, \ldots, t_{\mu+1}$. Also because α_k ends with `/` t_k, it can match with t_k only. If the regex matches t_k, it means

$$\Theta_{i,j} \texttt{/} t_k \in \mathcal{L}(\texttt{.*}\ \overline{\text{SEP}}\ \lambda\ \lambda\ \texttt{(}\ \Theta_{i,j}\ \texttt{)}\ \sigma_k\ \Lambda\ \texttt{/}\ \overline{\text{SEP}}\ \Lambda\ \texttt{/}\ t_k)$$

meaning

$$\Theta_{i,j} \in \mathcal{L}(\texttt{.*}\ \overline{\text{SEP}}\ \lambda\ \lambda\ \texttt{(}\ \Theta_{i,j}\ \texttt{)}\ \sigma_k\ \Lambda\ \texttt{/}\ \overline{\text{SEP}}\ \Lambda)$$

As explained in Section 5,

$$\mathcal{L}(\texttt{(}\ \Theta_{i,j}\ \texttt{)}) = \phi(x_{i+1,1}) \cdots \phi(x_{i+1,j}) = \phi(x_{i,3}) \cdots \phi(x_{i,j+2})$$

Therefore, the match happens if and only if $\sigma_k = \phi(x_{i,j+3})$. That is, t_k is deleted if and only if there exists a $\sigma_k = \phi(x_{i,j+3})$, and no marker files are deleted if there is no such σ_k, meaning $j \geq |w_i| - 2$. Whether deletion happens or not, the code block from line 9 is evaluated. This is evaluated to true creating a directory $\Theta_{i,j+1}$ if and only if there exists a deleted marker file t_k, because the k-th expression's `find` exits with non-zero status code only if the starting point $\Theta_{i,j}$ `/` t_k does not exist. The code block from line 14 is therefore evaluated when $j = |w_i| - 2$ for the first time for each i. By construction this turns the state into $\Theta_{i+1,0}$ (see the proof in Section 3 for details). Finally, because of the `,` operator, the line `-exec find` SEP `-type f -delete ;` is always evaluated, deleting all marker files, restoring the cleanness condition.

The third command (if the second command halts) will output $\Phi(w_t)$ as explained in the proof in Section 3. ◀

B Toy Examples

B.1 `find` + `mkdir` with back-references

We use the following 2-tag system instance as an example[4]. The expected output is `Hccccca`.

$$\begin{aligned} \Sigma &= \{\texttt{a}, \texttt{b}, \texttt{c}, H = \texttt{H}\} \\ P(\texttt{a}) &= \texttt{ccbaH} \\ P(\texttt{b}) &= \texttt{cca} \\ P(\texttt{c}) &= \texttt{cc} \\ w_1 &= \texttt{baa} \end{aligned}$$

Following shell script directly translates the construction in Section 3. If it is run, it outputs the encoded halting word `/ad/ac/ac/ac/ac/ac/ac/aa`.

[4] Taken from `https://en.wikipedia.org/wiki/Tag_system#Example:_A_simple_2-tag_illustration`

```
#!/bin/bash
TMP="$(mktemp -d)"
cd "$TMP" # For safety.
# Start of the main program

A=("aa" "ab" "ac" "ad")
sigma=(/"${A[0]}" /"${A[1]}" /"${A[2]}" /"${A[3]}")
eta="${sigma[3]}"
lambda="/[a-z][a-z]"
Lambda='\('"$lambda"'\)*'

pi=(
      "${sigma[2]}${sigma[2]}${sigma[1]}${sigma[0]}${sigma[3]}"
      "${sigma[2]}${sigma[2]}${sigma[0]}"
      "${sigma[2]}${sigma[2]}"
)
Phiw1="${sigma[1]}${sigma[0]}${sigma[0]}" # baa

alpha=(
   '.*_'"$lambda$lambda"'\('"$Lambda"'\)'"${sigma[0]}$Lambda"'/_\1'
   '.*_'"$lambda$lambda"'\('"$Lambda"'\)'"${sigma[1]}$Lambda"'/_\1'
   '.*_'"$lambda$lambda"'\('"$Lambda"'\)'"${sigma[2]}$Lambda"'/_\1'
   '.*_'"$lambda$lambda"'\('"$Lambda"'\)'"${sigma[3]}$Lambda"'/_\1'
)
beta=(
   '.*_'"${sigma[0]}$Lambda"'/_'"$Lambda"
   '.*_'"${sigma[1]}$Lambda"'/_'"$Lambda"
   '.*_'"${sigma[2]}$Lambda"'/_'"$Lambda"
)
gamma='.*_\(\|'"$lambda"'\|'"$eta$Lambda"'\)/_'

mkdir -p '_'"$Phiw1"'/_'
find _ -empty \( \
      -regex "$gamma" -quit -o \
      -regex "${alpha[0]}" -execdir mkdir {}"${sigma[0]}" \; -o \
      -regex "${alpha[1]}" -execdir mkdir {}"${sigma[1]}" \; -o \
      -regex "${alpha[2]}" -execdir mkdir {}"${sigma[2]}" \; -o \
      -regex "${alpha[3]}" -execdir mkdir {}"${sigma[3]}" \; -o \
      -regex "${beta[0]}" -execdir mkdir -p {}"${pi[0]}"/_ \; -o \
      -regex "${beta[1]}" -execdir mkdir -p {}"${pi[1]}"/_ \; -o \
      -regex "${beta[2]}" -execdir mkdir -p {}"${pi[2]}"/_ \; -o \
      -printf unreachable \
\)
find _ -depth ! -empty -name _ -execdir find _ ! -name _ -printf /%f \; -
    quit

# End of the main program
rm -rf "$TMP"
```

B.2 GitHub repository

Other toy examples are available at the author's GitHub repository `https://github.com/ogiekako/pub_find_mkdir`.