Losing Find

Will Thompson

May 15, 2008

Abstract

Most current desktop environments feature some kind of document indexing and searching technology, such as Quicksilver on the Mac and Tracker in the GNOME desktop. These are generally presented to the user in a dedicated search interface, distinct from the normal representation of the file system structure. As a result, each application needs to be modified in order to make use of this functionality; some applications have been, but many have not.

Modern Unix implementations allow filesystem drivers to be implemented in user space; that is, sections of the system's directory hierarchy can be provided by normal, unprivileged applications rather than by kernel modules. Using this, the results of queries against the document index can be represented as regular directories and files, allowing such queries to be used anywhere that a normal path can be used. A model for such a representation is given, along with FindFS, a working prototype implementation written in Haskell.

Contents

Motivation

The current state of the art for searching for files based on their attributes and contents from the Unix shell is to use the `find` and `grep` commands. Both of these commands present matching files as a list of file names, which is not a convenient format if you want to explore the results further. If you want to open every matching file in a single application, you can pass the `-exec` flag to `find` and have it run that application for you, but if want only to examine some of the matching files, or use different viewers for different files, you have to manually copy-and-paste the paths. Both commands also suffer from a more fundamental problem: they must examine every file in the relevant directory hierarchy in order to evaluate every query. While this is not a problem for searching small sets of files, the size of modern storage media make this a significant cost when searching many or all of your files, and it is wasted effort when indices of the relevant information already exist.

For an example, suppose you are a frequent lecturer, and you keep all of your lecture slides within a `slides` directory. After years of presenting on various topics, that directory is unmanageably large, so you would like to go through all slides older than one year and larger than two megabytes to determine whether to keep them for further use, or archive them into the `old-slides` directory. So, you issue the command:

```
[~]% find ~/slides -ctime 365 -size 2m
```

The output is many, many pages of file names. If you wanted to archive them all, you could enter:

```
[~]% find ~/slides -ctime 365 -size 2m -exec mv {} ~/old-slides
```

or pipe the ontput of the `find` command to an invocation of `xargs`. But since you want to archive only some of those files, you are stuck with wading through a list of files one by one, copy-and-pasting the file name into your viewer. If you could view the results of this search as if it were just another directory, you could use your shell's tab completion functionality to enter the names of files, or even view the set of matching files in your graphical file browser (which can probably sort by age or size, but not by both).

While `grep` is adequate for searching the contents of text files, other tools must be used to search the metadata in more elaborate file types, such as audio and images. Suppose you need to find a number of images at most 200 pixels wide by 300 pixels high, to include in a document you are writing. To find such images, you could open your photograph library application of choice and perform the search there (assuming that it supports such searches), then copy the images back into your document editor. But if you have a large volume of photographs not yet imported into the library, as does this author, such a search would not find all relevant results. A different approach is to search using the document indexing and retrieval search on your system, which circumvents the requirement to manually catalogue your photographs ahead of time. However, in both cases you have had to leave your document editor to launch the querying interface, rather than choosing the editor's "Insert image..." option and navigating to the relevant files, since you did not know ahead of time what those files are, and where exactly they are stored.

Similarly, suppose you have a scene classification application which, given an image, attempts to calculate the number of people in that image. The calculation is expensive, so you would like to plug it into your document indexing system so that the result can be computed once per image and then cached for as long as the image is unchanged. However, you would like to be able to use this information from within various applications on your system without having to modify each application to be aware of this new knowledge, rather than needing to use the dedicated searching interface provided by the indexing system.

All of the previous examples have involved searching for files with particular properties, rather than for directories. Suppose that you keep all of your software projects in various subdirectories of `~/source` (perhaps you keep your "Foo" project in `~/source/personal/foo`) and want to see which of them are under version control with the Darcs version control system. To find such directories, you can look for directories which contain subdirectories named `_darcs`. Doing this with `find` is rather counter-intuitive, as you must search for the `_darcs` directories, and then print to the terminal a list of the parent directory of each match:

```
[~]% find ~/source -type d -name _darcs -printf '%h\n'
```

So, whether we use `find` and `grep` or a dedicated desktop search UI, we suffer from being unable to pose queries and manipulate their results from within arbitrary applications. The proposed solution is to represent queries against a document index as virtual directories within the normal filesystem hierarchy, allowing them to be posed and viewed everywhere that standard directories can. FindFS allows any application to take advantage of the document index—and new facts about files it may learn to store in the future—without modification. It can also serve to replace many uses of `find` and `grep`: all of the standard shell utilities can be used on query results, without having to resort to `xargs` and copy-and-pasting.

This paper proposes a way to expose the querying functionality of a desktop search tool as a virtual filesystem, named FindFS, so that queries can be posed within unmodified applications and their results browsed and manipulated using standard shell commands—or indeed a graphical file browser. Queries are represented by virtual directories, overlaid onto a view of the user's home directory and distinguished from regular directories by a symbol, ⊗. The majority of the queries given above can be posed within FindFS. For a user Alice, the search for old files within the directory `~/slides` would be represented by the directory:

```
/find/alice/slides/⊗(File:Modified <= 1 year ago)/⊗(File:Size >= 2 mb)/
```

To search the whole of `/home/alice` for images within the given dimensions, you would examine the directory:

```
/find/alice/⊗((Image:Width <= 200) and (Image:Height <= 300))/
```

Having extended Tracker to run the hypothetical scene classifier over all indexed images, you would be able to treat the following as a constantly-updated directory of photographs in `~/Pictures` depicting groups of at least three people:

```
/find/alice/Pictures/⊗(Image:People >= 3)/
```

The model presented does not allow queries based on properties of directories, so it is not possible to represent the given example search for version-controlled projects with FindFS. A possible extension to the model to allow such queries will be briefly discussed towards the end of this paper.

Design

Conceptual model

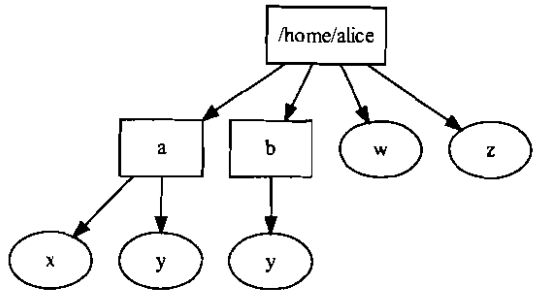

Figure 1: The contents of a user's home directory, for the following examples

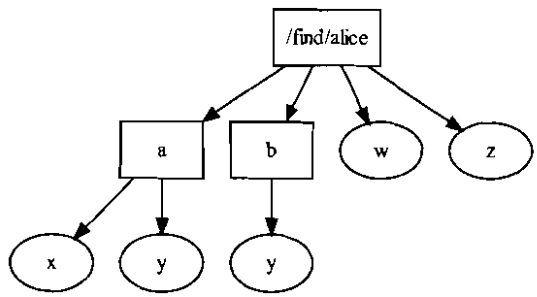

Figure 2: The full contents of Alice's home directory, mirrored under `/find`.

FindFS is mounted at `/find`, and contains a directory entry for each indexed directory. Typically, the user's home directory is indexed, so for a user whose data lives in `/home/alice`, `/find` will contain a directory `alice`. In the absence of a query, these behave as mirrored views of the underlying directories—with the same directories (represented by rectangles in diagrams) and files (ovals)—and can be used in exactly the same way. Each file is a **proxy** for the real file; modifying the contents of the proxy will also modify the real file, and vice-versa.

Queries against the corpus are *conditions on files* (they cannot match directories in this model). In the filesystem representation, they take the form of virtual directories (denoted by diamonds) within the replicated directory hierarchy, expressed in a notation to be defined later. These virtual directories do not appear in listings of their parent directory, but spring into temporary existence as and when they are accessed. (I will refer to this "invisible unless directly addressed" behaviour by describing the directory as **transient**, denoted by dotted lines.) The tree below such a query directory is based on the tree below the parent directory, but restricted to files matching the query and the directories leading to such files.

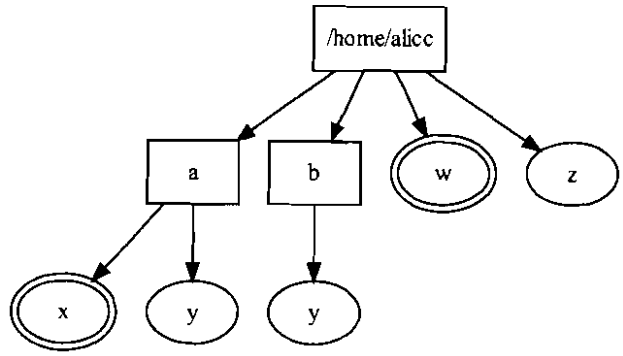

Figure 3: Files in `/home/alice` matched by a query q.

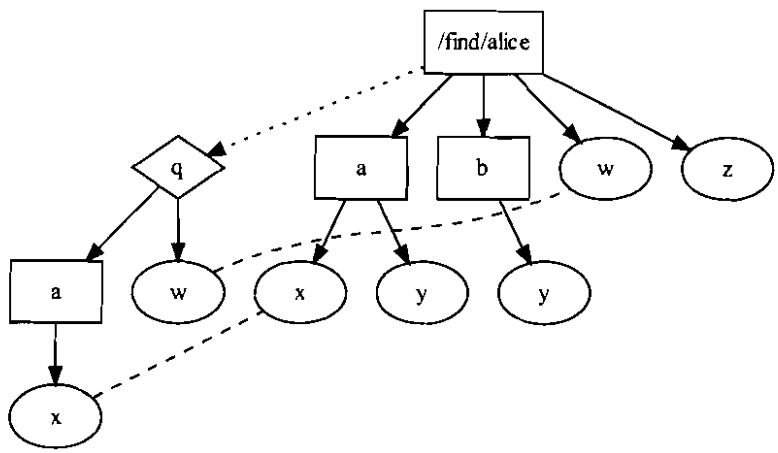

Figure 4: Evaluating the query q at `/find/alice`.

Each such file is the same inode as the file beneath the parent directory, and as such is once again a proxy for the underlying file.

For example, take a query q which matches two files of the example hierarchy namely `/home/alice/a/x` and `/home/alice/w` (indicated by double ovals in Figure 3). Then, as shown by Figure 4, `/find/alice/`q is a directory containing a file called `w`, since `/home/alice/w` is matched by q, and a directory named `a`, corresponding to `/home/alice/a`. In turn, `/find/alice/`q`/a` contains a single file, `x`, since `/home/alice/a/x` matches q. `/find/alice/`q`/w` is a proxy for `/home/alice/w`—they have the same contents and attributes, and modifying one will modify the other—as `/find/alice/`q`/a/x` is for `/home/alice/a/x`. (The directory `b` does not have a corresponding entry in `/find/alice/`q since none of its contents match q, and `z` does not appear as it does not match q.)

Queries can also be applied at paths strictly below the root of the mirrored structure in the same way. For

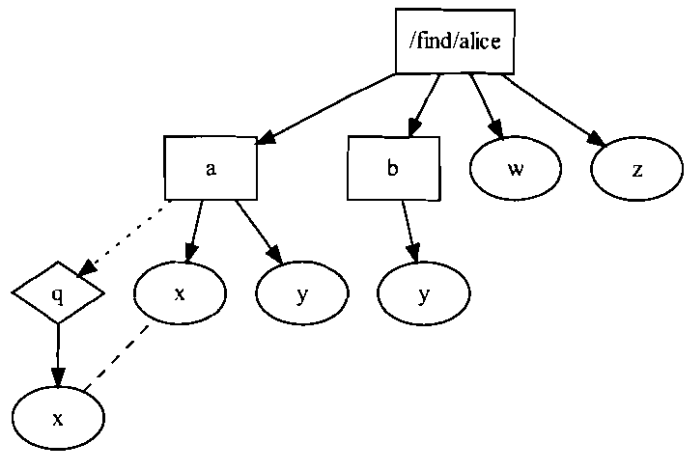

Figure 5: Evaluating the same query q at `/find/alice/a`.

the same query q, `/find/alice/a/`q is a directory containing only a file `x` proxying `/home/alice/a/x` (as shown in Figure 5).

From this example, we see that query components in a path commute with "real" components, in the sense that if the directory `foo/bar` exists in the underlying directory structure, then the contents of `foo/bar/`r, `foo/`r`/bar` and r`/foo/bar` are identical for any query r (as shown in Figure 6).

It might seem strange to retain the underlying directory structure when presenting a query's results as a virtual directory, rather than representing the query by a directory containing all matching files. But that representation runs into problem when a query matches two files with the same base name (that is, the final component of their paths). Consider a query p matching both `a/y` and `b/y` in the running example. Evaluating it beneath either directory would not be problematic were we to flatten all results into one virtual directory. However, if it were to be evaluated at the root, it would need to contain two files named `y`, which is impossible. This problem does not arise when the relevant branches of the directory hierarchy are preserved, as two files with the same name can only occur within different directories (as illustrated in Figure 7).

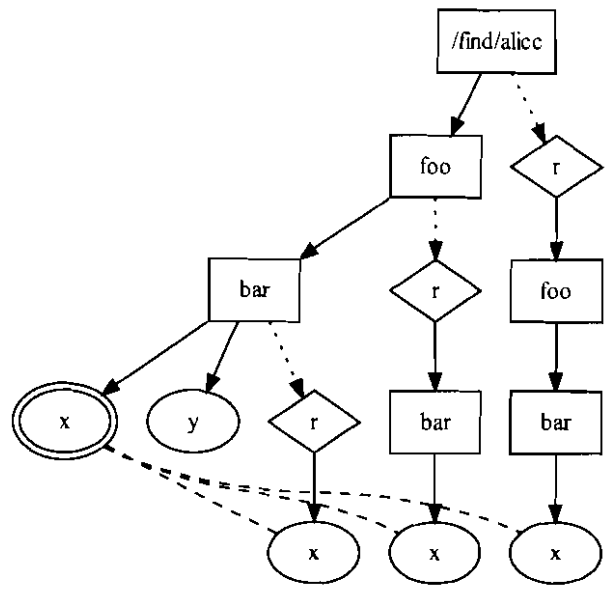

Figure 6: Showing how a query r matching `foo/bar/x` commutes with `foo` and `bar`.

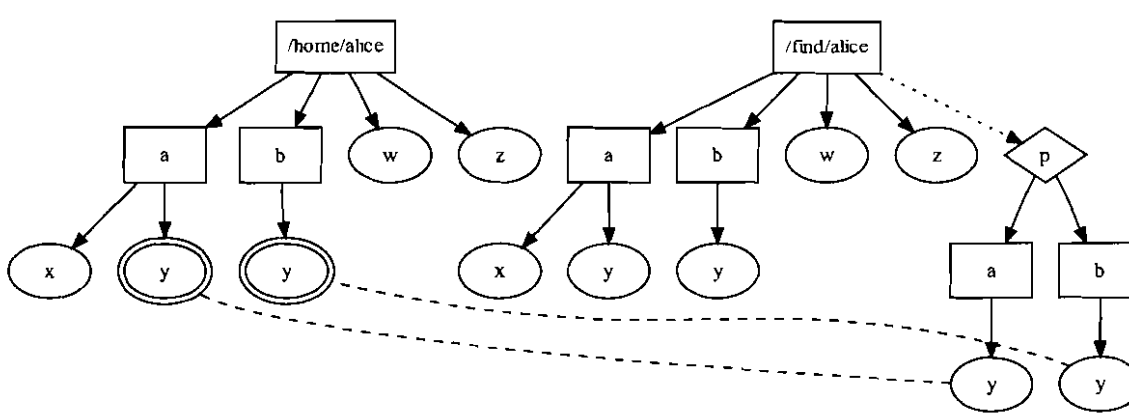

Figure 7: The results of a query p which matches several files sharing the basename y.

Definitions and axioms

Consider a general path $P \stackrel{\text{def}}{=} c_1/\ldots/c_n$ relative to a corpus within `/find` for c_i components of the path separated by forward slashes in the normal way. (This would correspond to `/find/alice/`$c_1/\ldots/c_n$ in the above examples.)

- If none of the c_i are queries, then we say P is an **unfiltered path**; otherwise, if any of the c_i are queries, P is said to be a **filtered path** or a **query path.**
- If q is a query and x is any other path component, then the paths $P_1/q/x/P_2$ and $P_1/x/q/P_2$ are equivalent. That is, **query components commute with other path components.**
- Thus, P can be **normalized** to a path $P' = u_1/\ldots/u_k/q_1/\ldots/q_m$, where $U \stackrel{\text{def}}{=} u_1/\ldots/u_k$ is query-free and $Q \stackrel{\text{def}}{=} q_1/\ldots/q_m$ consists entirely of query components, by commuting all query components to the end of the path. So in normal form, a path is a series of non-query components, optionally followed by a series of query components.
- The **underlying path** of P is U, relative to the location at which the corpus in question is actually stored (so `/home/alice/`U in the above examples). The query $\overline{Q} \stackrel{\text{def}}{=} \bigwedge_i q_i$ is the **filter** of P. (If P is unfiltered, then $\overline{Q}$ will be identically true, matching every file.)

Now the existence and nature of the FindFS node at P is determined as follows:

- Normalize P, and examine its underlying path.
- If there is no object (file, directory, etc.) at the underlying path of P, then there is no node at P.
- If the object at the underlying path of P is a file f, then:
  - if f matches $\overline{Q}$, then the node at P is a proxy for f.
  - if not, then there is no node at P.
- If the object at the underlying path is a directory d, then P is a directory containing proxies for those files in d matching $\overline{Q}$, and a directory entry for each directory in d with a descendant file matching $\overline{Q}$.

We could insist that the user express all paths in normal form, but we do not for several reasons. Firstly, if a virtual directory $\ldots/q$ contains a directory `foo` whose contents we want to examine, it would inconvenience the user to require that `foo` be added to the path before q rather than allowing it to be appended to the path. Secondly, we will later introduce the concept of *reifying* a path, where the position of query components within a path may in fact affect the results. That said, the FindFS implementation does effectively normalize paths in the course of evaluating them, so the concept has a practical purpose.

Query notation

The Tracker document indexing system, which is used by the FindFS implementation, indexes various aspects of the files under its purview. The size, creation date, etc. of all files is monitored; in addition, Tracker attempts to determine and record the type of each file, and in some cases examine type-specific metadata. For instance, the artist and album metadata for MP3 audio files is indexed, as are the title, author and contents of text-like files such as PDF documents. Queries can be posed in terms of conjunctions and disjunctions of propositions based on these attributes; for example, you could search for files satisfying *(last modified more than one year ago or larger than two megabytes) and (is an image).*

For the purposes of FindFS, these conditions must be stated in a notation meeting the constraints of Unix directory names, which can be distinguished from ordinary path components. So, queries are prefixed with a symbol, ⊗. The predicates themselves are expressed in a straightforward *attribute-comparison-value* fashion; the example just given is written:

```
⊗((File:Modified < 1 year ago) or (File:Size > 2 mb)) and (File:Type == "image")
```

Since ultimately we will take the conjunction of all query components in a path, this query could also be expressed as nested directories:

```
⊗(File:Modified < 1 year ago) or (File:Size > 2 mb)/⊗(File:Type == "image")
```

As a special case, the condition *document contains the string "foo"* is written as ⊙`foo`.

Meanings of filesystem operations within FindFS

In order for FindFS to be useful, it has to behave exactly as the underlying filesystem would do when dealing with unfiltered paths, and behave predictably and coherently when filtered paths are in use. The majority of operations have obvious interpretations; for example, reading the contents of a file within a filtered path should just perform the same read on the file in its true location behind the scenes. Similarly, modifying the permissions and contents of files viewed through a filtered path makes sense, although of course modifying a file might make it no longer match the query and thus vanish from the virtual directory you're looking at!

More problematic are operations that should modify the contents of a directory. Recall the filtered directory of all images smaller than 200×300 pixels:

```
/find/alice/⊗((Image:Width <= 200) and (Image:Height <= 300))/
```

Creating a new (empty) file `foo` below this directory would be problematic for a number of reasons. There might already exist a file `foo` in the underlying directory not matching the query, so the operation would have to fail with a "File exists" error despite the clashing file not being visible. More importantly, the newly-created `foo` would be empty, and so would not match the query and would not be shown in the filtered directory! Thus, we forbid creating new files in filtered directories. This might prove frustrating in cases where the newly created file will ultimately match the relevant queries, or if an editor attempts to create a temporary file in the same directory as a file you are editing, but I can see no straightforward way around it.

On the other hand, deleting files from a filtered directory is permitted, and is interpreted as deleting the files from the underlying directory. To justify this behaviour, consider again the virtual directory of large, old slides:

```
/find/alice/slides/⊗(File:Modified <= 1 year ago)/⊗(File:Size >= 2 mb)/
```

Looking over these slides, you might decide that some are obsolete and want to permanently delete them. It would be annoying to have to manually find the underlying file in order to delete it, and is in keeping with the proxying behaviour of files to delete the underlying file when the proxy is deleted.

Since we permit deleting files from any directory (filtered or not), and permit creating files in unfiltered directories, then it follows that we should allow files to be moved from filtered directories to unfiltered ones. Continuing the example, you may instead want to move all old slides of any size to an archive directory:

```
[/find/alice]% mv  slides/'⊗(File:Modified <= 1 year ago)'/*  old-slides/
```

The *rename*(*from*, *to*) system call—nsed both for renaming files within a directory and for moving them between directories on the same filesystem—is specified to do nothing if *from* and *to* are the same path. FindFS generalizes this: if *from* is a filtered path, and *to* is an unfiltered path identical to the underlying

path of *from*, then *rename*(*from*, *to*) is a successful no-op. We do not allow moving files into a filtered directory, just as creating new files in a filtered directory is forbidden.

Because of the manner in which directories are filtered, it turns out that deleting directories below a filtered path makes sense when the directory is empty, which is a precondition for deleting a directory on any filesystem. Consider the query q from Figure 3, which matches no files in the directory `/home/alice/b`. Listing `/find/alice/`q`/b` will show it to be an empty directory, so we would expect to be able to delete that directory. But the contents of `/find/alice/`q does not include `b`, since no files beneath it match q, so we can interpret `rmdir /find/alice/`q`/b` by doing nothing and returning success! In general, calling `rmdir` on a filtered directory path ending in a non-query component succeeds if and only if the filtered directory is empty; if the underlying directory is also empty, it is removed, but if not it remains untouched.

Persistence of queries with `mkdir`

Because every valid query is a subdirectory of any directory viewed in FindFS, it does not make sense to have all possible queries appear when calling `ls` on a directory: you would have to include infinitely many queries in the listing! But you might want a commonly-run query to appear in certain directory listings. We can attach this behaviour to calling `mkdir` and `rmdir` on filtered paths: calling `mkdir` on a filtered path succeeds (providing the underlying path exists), and causes the query components of the path to be in a sense **reified**—they will appear in subsequent listings of their parent directory. Calling `rmdir` on a filtered path that has previously being reified undoes this effect; that is, the query is no longer reified.

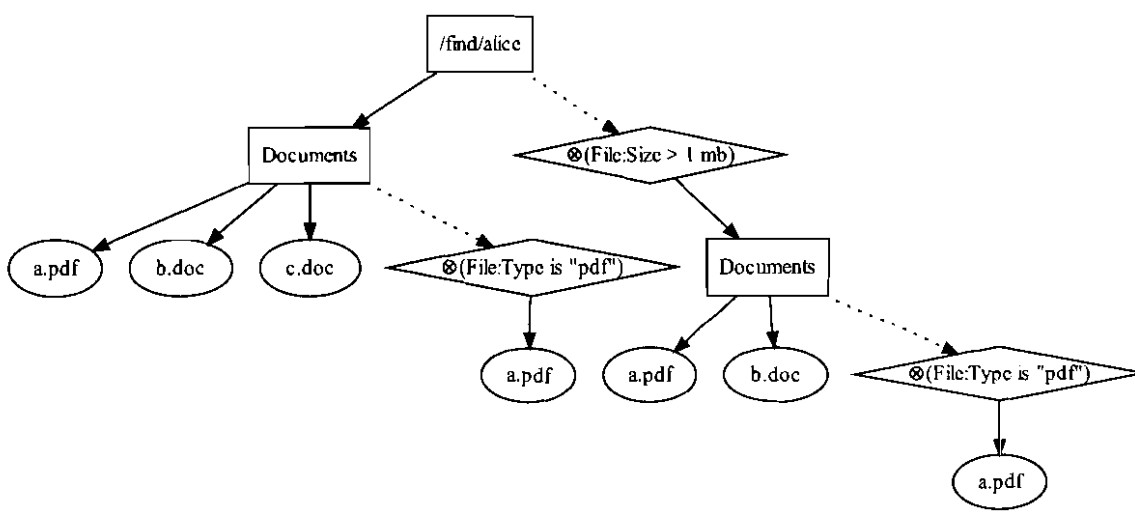

Figure 8: Before calling mkdir on `/find/alice/⊗(File:Size > 1 mb)/Documents/⊗(File:Type == "pdf")`

For example, consider calling `mkdir` on the path `/find/alice/⊗(File:Size > 1 mb)/Documents/⊗(File:Type == "pdf")`. Thereafter, listing the contents of `/find/alice` will return a directory entry `⊗(File:Size > 1 mb)` alongside the other contents of the directory, and the contents of `/find/alice/⊗(File:Size > 1 mb)/Documents` will include an entry `⊗(File:Type == "pdf")`. As seen in Figure 9, listing `/find/alice/Documents` will not include the `⊗(File:Type == "pdf")` sub-directory, since the former was not part of the reified path.

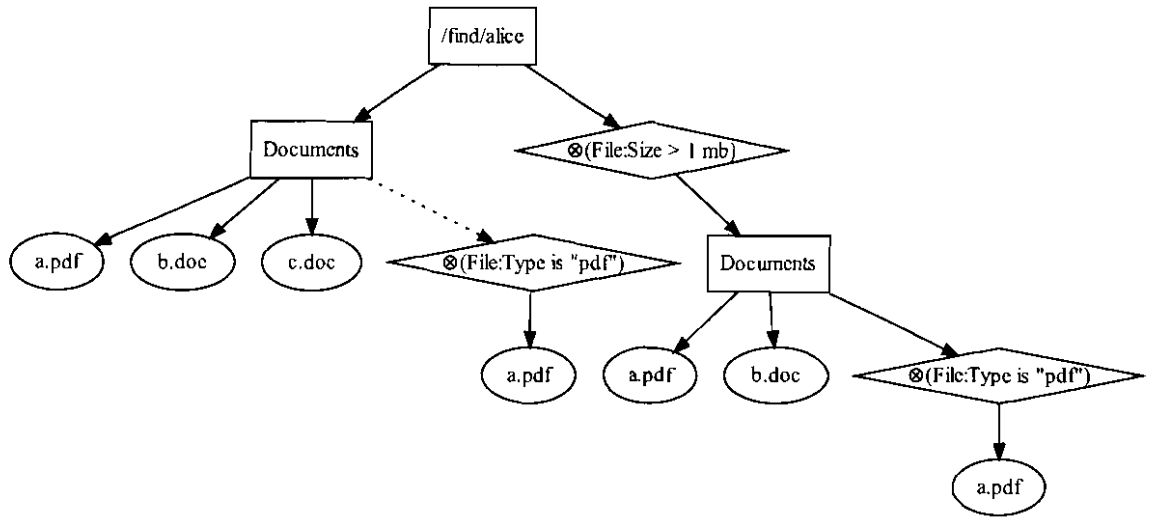

Figure 9: After calling mkdir on `/find/alice/⊗(File:Size > 1 mb)/Documents/⊗(File:Type == "pdf")`

As mentioned, this causes the position of query components within a path to affect the results of examining that path. In the presence of reified paths, the normalized form of a path is not equivalent to the original path if the original path is the prefix of a query component of at least one reified path. Indeed, they will differ precisely by those query components; in the above examples, if we normalize `⊗(File:Size > 1 mb)/Documents` before evaluating it, the implementation must remember to add the `⊗(type == "pdf")` entry to the results. In all other cases, a path can be normalized without changing its meaning.

Implementation

FindFS is implemented as a Haskell application, acting as a filesystem through the Fuse library and Linux kernel module. It delegates querying and indexing to the Tracker daemon.

Fuse

The Haskell interface to Fuse (which is a very thin wrapper around the C API) provides a record type (see Listing 1) with fields for each of the operations a filesystem may support. The application fills in as many of these fields as it can, then passes the record to the `fuseMain` function, which provides the main loop of the application. Each field is an action in the `IO` monad, which is invoked with the relevant parameters by the main loop as applications manipulate the mounted filesystem. Since multiple applications can access the filesystem concurrently, several actions may be called at once in different threads; it is up to the application to ensure that this is safe.

The Fuse API imposes no particular internal representation for the filesystem hierarchy on the application; all paths are provided as strings, relative to the mount point. So if a Fuse-based filesystem is mounted at `/mnt/misc`, and an application tries to make a directory `bar` inside `/mnt/misc/foo`, then the `fuseCreateDirectory` operation will be passed `/foo/bar` as the `FilePath` argument. It is up to the application to transform this path into its structured representation as necessary.

For operations examining or modifying a file's contents, the application can represent a file handle by an arbitrary object of its choosing, which will be passed as an argument to future operations on that file. FindFS takes advantage of this to represent proxied files as a standard Haskell `Handle` opened on the underlying file, and then simply delegates all subsequent operations to that handle via the usual Haskell file IO actions. (The C API for Fuse allows the application to instruct the kernel to perform this delegation directly; in effect, the application passes the real file handle back to Fuse, along with a flag specifying that all future operations on the handle—apart from closing it—should be performed directly on the real file handle, bypassing Fuse entirely. This dramatically improves performance, as it avoids making a large number of context switches to and from the filesystem process when performing file IO. However, the functionality is not currently exposed in the Haskell API, so FindFS does not make use of it.)

Tracker

Tracker is an indexing and document retrieval service for the Linux desktop. It knows how to extract metadata from most common types of file, and allows that metadata to be searched in various ways. Applications communicate with it via the D-Bus inter-process communication framework, or using a C library that wraps the D-Bus APIs with a pair of C APIs, one blocking and one asynchronous. While a Haskell library for D-Bus exists, it is in a poor state, so the C library was used to communicate with Tracker; since all Fuse operations are execnted in their own thread, only the blocking API was wrapped for use in Haskell.

Listing 1: Some of the members of the FuseOperations record

```
data FuseOperations fh = FuseOperations
  {
    fuseGetFileStat :: FilePath → IO (Either Errno FileStat),
    fuseCreateDirectory :: FilePath → FileMode → IO Errno,

    fuseOpen :: FilePath → OpenMode → OpenFileFlags → IO (Either Errno fh),
    fuseRead :: FilePath → fh → ByteCount → FileOffset → IO (Either Errno ByteString),
    fuseWrite :: FilePath → fh → ByteString → FileOffset → IO (Either Errno ByteCount),

    -- many more operations elided
  }
```

Listing 2: Key elements of Haskell API for Tracker

```
module System.Tracker where

-- A connection to the Tracker daemon, which can execute one query at a time.
newtype Client = ...

-- Creates a new connection to the Tracker daemon.
connect :: IO Client

-- In fact, this function has many more arguments, which have been omitted for clarity.
searchQuery :: Maybe String      -- optional search terms for a full text search
            → Maybe Condition    -- an optional query condition
            → Int                -- offset from start of list of hits
            → Int                -- maximum number of hits to return
            → Client
            → IO [FilePath]      -- paths to matching files

newtype Condition = Condition { unCondition :: Clause }

data Clause = Atom Operator Property Value
            | And [Clause] | Or [Clause] | Not Clause

data Operator = Equals | GreaterThan | GreaterEqual | LessThan | LessEqual
              | Contains | StartsWith
newtype Property = Property String
data Value = IntV Int | DateV String | StringV String | FloatV Double
```

The function most relevant to FindFS is `searchQuery`, which searches for files matching a full-text search (which correspond to ⊙-prefixed queries in FindFS's representation) and/or a predicate on files' metadata (corresponding to the more general ⊗-prefixed queries). Valid property names include `File:Mime` (the file's MIME type, such as `text/html`), `Doc:PageCount`, `Image:Date`, and many others, largely based on the `freedesktop.org` file metadata specification[1] .

It is an unfortunate limitation of Tracker that, despite the fact that files' full paths are of course indexed, it does not seem to be possible to limit queries to descendants of a particular directory (and the maintainer seems opposed to adding such functionality). As a result, FindFS has to filter the results to the relevant path itself, possibly retrieving a large number of irrelevant results in the process. Of course, whether FindFS or Tracker does this filtering ultimately makes no difference to the results; on the other hand, the SQLite database underlying Tracker is in a position to do a much quicker job of pruning irrelevant matches than FindFS, which has no indices to assist it and must fetch each result over a relatively slow IPC system. This is mitigated somewhat by the offset and limit parameters of `searchQuery`: if the first n results tell us everything we need to know, we need request no more from the daemon.

Tracker's query language and D-Bus interface is set to change in the relatively short term to match the XESAM[2] standard. When this occurs, migrating FindFS to the new interfaces would allow it to work with other indexers for the Linux desktop without further modification.

Implementation of FindFS

In the current implementation, FindFS only supports mirroring a single directory, namely the user's home directory. When invoked, it determines the path from the `$HOME` environment variable, and mounts itself at `/find`, presenting a mirror of `$HOME` in a directory sharing its baseuame. (So as in the above examples, if `$HOME` is `/home/alice`, then `/find/alice` will be the root of the mirror.)

All Fuse operations return a Unix `Errno` value on failure. When a successful operation needs to a value of type a, its return type is `Either Errno a`, using the standard mnemonic that `Right` signifies success. Other operations need produce no value on success, so the return type of the callback is `Errno`, with the

[1] `http://freedesktop.org/wiki/Specifications/shared-filemetadata-spec`.
[2] `http://xesam.org/`

Listing 3: FindFS's monad transformer stack

```
data FindEnv = FindEnv { envCorpus :: FilePath, envState :: MVar FindState }
data FindState = FindState { reifiedPaths :: Set FilePath }

type Find = ReaderT FindEnv (ErrorT Errno IO)
type StatefulFind = StateT FindState Find

-- Runs a stateful action in the Find monad.
statefully :: StatefulFind a → Find a
statefully act = do
    stateVar ← asks envState -- Retrieve the state's box from the environment

    -- Remove the current state from the box; further attempts to read the box's
    -- contents will block until a new value is put in.
    s ← liftIO (takeMVar stateVar)

    -- Run the supplied action starting from the current state, yielding a
    -- result and a possibly-modified state.
    (result, s') ← runStateT act s

    -- Place the modified state back into the shared box, allowing threads
    -- blocking on the state to continue.
    liftIO (putMVar stateVar s')

    return result
```

constant `eOK = Errno 0` being the "error" code signifying success. This matches the underlying C Fuse API, but the inconsistency is clumsy, so FindFS wraps the latter style of operation to have return type `Either Errno ()`.

Monadic environment

Rather than operating directly within the `IO` monad, FindFS code is embedded in a stack of of monad transformers around `IO`, providing clean access to early termination with an error code and appropriate immutable and mutable state for the application. The stack seen in Listing 3 is built up as follows:

- Working directly within monadic actions returning `IO (Either Errno a)` is inconvenient. Instead, FindFS actions operate within `ErrorT Errno IO a`, allowing computations to short-circuit on error by calling `throwError` with the relevant error code, and otherwise yield a value of type `a` on success.
- The root directory of the underlying corpus will be needed at various points. Rather than explicitly passing around a string, FindFS uses the `ReaderT` monad transformer to allow access to a global immutable value of type `FindEnv`. So we define the monad `Find a` as an alias for `ReaderT FindEnv (ErrorT Errno IO) a`.
- Some mutable state is needed; in particular, the set of reified query paths needs to be tracked, and we need to ensure that modifications to that set are made atomically. We can fulfil both requirements by keeping a pointer to a mutable cell in the immutable environment; using the `MVar` pointer-like structure provides synchronized access and updates to that cell. To simplify code using the state, FindFS has a wrapper function—`statefully`—exposing it as a `StateT`-transformed version of the `Find a` monad, so that modifications can be made through the standard `MonadState` actions. Thanks to the synchronization features of `MVar`, such stateful actions are atomic with respect to other actions manipulating the shared state, while allowing actions that don't need to inspect the state to continue unimpeded.

In order to invoke a `Find a` action from a Fuse callback, all of the `FuseOperations` members are wrapped with a function calling `(runErrorT . runReaderT env)` on the action, having constructed the appropriate environment.

Listing 4: Delegating operations to the underlying filesystem

```
type Component = ...
type FindPath = [Component]

data Node = Node { nodeName :: Component
                 , nodeVirtualPath :: FindPath
                 , nodeUnderlyingPath :: FilePath
                 , nodeTextQuery :: Maybe String
                 , nodeCondition :: Maybe Condition
                 }

-- Raises "eNOENT" (file not found) if the path cannot be parsed
parsePath :: FilePath → Find Node

-- Calls the given primitive IO operation on the underlying path of a FindFS
-- path, throwing "eNOENT" if the file does not exist or does not match the query.
evaluateAtPath :: (FilePath → IO a) → FilePath → Find a

-- Calls the given primitive IO operation on the underlying path of a FindFS
-- path, throwing "eACCES" (operation not permitted) if the FindFS path is filtered.
evaluateAtUnfilteredPath :: (FilePath → IO a) → FilePath → Find a
```

Evaluation of filesystem operations

Having parsed a path supplied by Fuse into the underlying path and the query conditions, it remains to evaluate the requested operation against it. Many of the operations on a file—such as `fuseSetFileMode`, which corresponds to the **chmod** command at the Unix shell, changing the permissions of a file—follow the same strategy: if the underlying file exists, and matches the query (if present), then simply call the standard `setFileMode` function on the underlying file; otherwise, fail with "File not found". We can abstract this pattern into a function `evaluateAtPath`, which takes a primitive `IO` operation and a virtual path and does the necessary parsing, querying and ultimate evaluation, significantly reducing the boilerplate necessary to implement many of the Fuse callbacks. Similarly, operations like `fuseCreateDevice` which may only succeed in unfiltered paths (as discussed previously) can be implemented with `evaluateAtUnfilteredPath`, which fails if the supplied virtual path is filtered and otherwise delegates the given `IO` action down to the underlying file.

Caching query results

Compared to operations on unfiltered paths, listing the contents of a filtered directory or acting upon a file below a query is a relatively slow process. In the current implementation, no qnery results are ever cached by FindFS, so listing the contents of the same filtered directory twice in succession will involve FindFS rerunning the necessary queries and recomputing the contents, even if (as is likely) the contents are unchanged. Given a way to monitor files and directories for changes, FindFS could cache the results of recent queries until it receives a signal that the cache has been invalidated, at which point it could decide whether to npdate the cache or to drop the cached entry based on how recently the results were needed. One way that this could be implemented is with the Linux kernel's `inotify` interface, through which applications can register to be notified of changes to particular files and directories. However, only changes to a directory's immediate children are signalled by `inotify`, so FindFS would have to recursively register its interest in everything below the relevant directory in order to receive the required notifications, which would be error-prone and unwieldy.

In the future, it will be possible for applications to inform Tracker that a particular query should be *live*: after returning the requested results, the Tracker daemon will send a signal to the application whenever the contents of the results change, until the application instructs it to stop. This would be ideal for FindFS, but unfortunately is not yet implemented. When it is, FindFS could cache query results, updating the cache when signalled by Tracker, and garbage-collect old queries after a certain period of time or when too many queries are live. Furthermore, FindFS could take the fact that a filtered

path has being reified as a hint to cache the query results ahead of time, so that even the first use of such paths in a session is speedy.

Lazy evaluation of queries

One of the great strengths of the Haskell language is its built-in support for lazy evaluation of values. It would seem sensible to treat the list of paths matching a given query as a lazy list, so that if FindFS is checking that a particular file matches, it can retrieve only as many results as are needed. However, repeated calls to the `searchQuery` action must be used to fetch segments of the result set, which are then concatenated to yield the full list; since this is an `IO` action, and such actions imposed a sequenced order of execution, the list will be fully evaluated before it is ever consumed. To work around this, we use the `unsafeInterleaveIO :: IO a → IO a` function to delay the execution of each action until its section of the result list is needed. The term "unsafe" signify that some of the ordinary ordering guarantees about `IO` actions is lost, and it is up to the programmer to ensure this does not lead to erroneous behaviour. In this situation, it does not, as the order of calls to `searchQuery` relative to one another is maintained.

Using this technique, FindFS contains a function

```
searchAll :: Maybe String → Maybe Condition → IO [FilePath]
```

which returns a lazy list of results, fetched 100 at a time on demand (a figure arrived at by trying various orders of magnitude and observing how long various queries took to be evaluated). So when checking whether a particular file matches a query, FindFS can just call the standard list function `elem` on the result of `searchAll`; behind the scenes, once the desired path is found no more results will be fetched. This reduces the cost of FindFS having to filter the results list itself, rather than having some way to instruct Tracker to restrict the set of results returned based on a path.

Sadly, in the case of listing a directory's contents, there is currently no benefit to handing a lazy list to Fuse, as the code marshalling the result of `fuseReadDirectory` back to the C API consumes the entire list immediately. While this is acceptable if the application reading the directory's contents wants the complete list (as does `ls`, for instance), in some cases the application will only need a few entries. The C API for Fuse supports returning only the requested number of entries; since `readdir(3)` operates on a fixed snapshot of the directory, it would be reasonable for Fuse's Haskell API to support consuming the returned list lazily. (Since this is an uncommon case, this modification was not made.)

Future work

Moving files into a filtered directory

As previously noted, it does not make sense in general to permit moving files into a filtered directory, as the files may well not match the filter. In some limited cases, FindFS might be able to observe that the file being moved does in fact match and hence permit the move. A slightly more interesting situation is where a file's metadata could be *modified* to match the filter; in this case, we could allow files to be moved into filtered directories in a number of useful cases.

For example, suppose that Alice has a file `My_Monkey.mp3` downloaded from a particular musician's web site, but that the file lacks the MP3 metadata tag specifying the artist. One could imagine a command `metamv` which would allow Alice to run:

```
[/find/alice]% metamv  My-Monkey.mp3  Music/'⊗(Audio:Artist == "Jonathan Coulton")'/
```

This command would cause FindFS to check whether the file matches the query, find that it doesn't, and use an MP3 manipulation library to add the *artist* metadata to the file before moving it into the underlying directory, `Music`. Of course, this operation would fail if FindFS could not modify the file to match the filter without damaging the file; for instance, an image cannot be made to have particular dimensions without scaling or cropping it. (It seems sensible to suggesting making this a separate command, rather than adding it to the standard *rename* operation, since the behaviour might be surprising if you were not expecting it.)

Extending the query model to allow conditions on directories

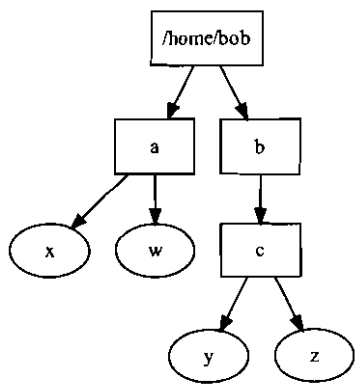

Figure 10: Bob's home directory; `b` has one child, while `a` and `c` have two.

The query model presented and implemented only allows predicates to contain conditions on files; as we have already noted, this is insufficient for certain types of query. Introducing conditions on directories is not straightforward, as they will not in general commute with other path components.

To illustrate this, let q denote the query *has exactly one child*, and consider Bob's home directory (Figure 10). We would expect `/find/bob/`q to contain those entries in `/find/bob` matching q, and directories in `/find/bob` with a descendant matching q, just as we do when working with conditions on files. So

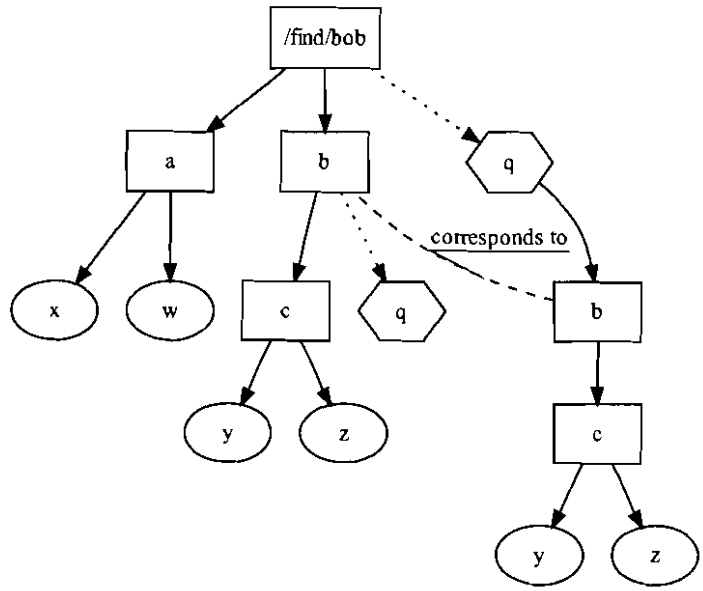

Figure 11: Evaluating $q \stackrel{\text{def}}{=}$ *contains only one child* at `/find/bob` and at `/find/bob/b`.

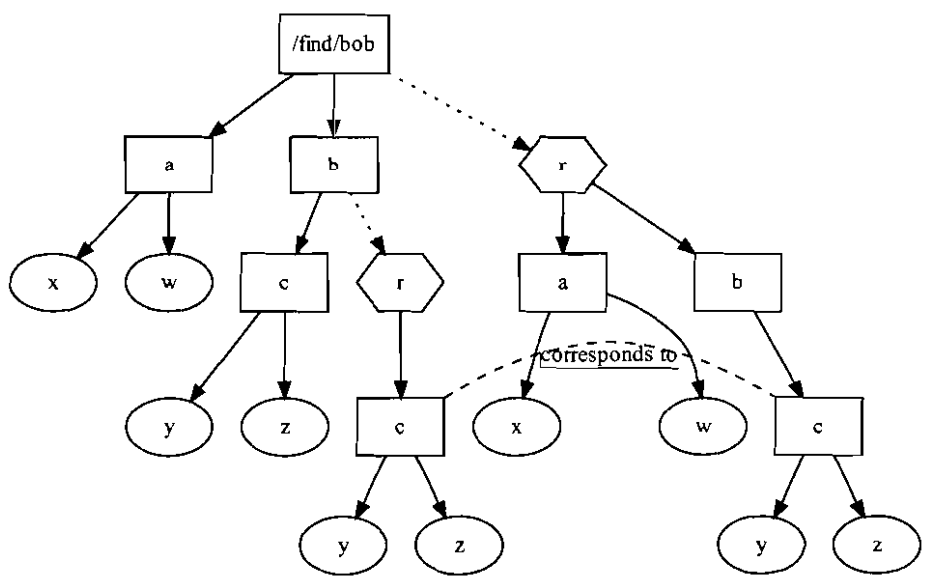

Figure 12: Evaluating $r \stackrel{\text{def}}{=}$ *contains two children* at `/find/bob` and at `/find/bob/b`.

`/find/bob/`q should contain `b`, as shown in Figure 11. Below q`/b`, we consider q to have been satisfied, and thus q`/b` contains a full proxy of `b/c`.

However, `/find/bob/b/`q is a different story. `b` contains no directories matching q, and no directories with descendants matching q; thus, `/find/bob/b/`q is empty, unlike `/find/bob/`q`/b`. This example shows that q does *not* commute with `b`, because `b` *satisfies* q.

Now, let r denote the query *has at least two children*. As depicted in Figure 12, this query does commute with `b`, as it is not `b` but a child of `b` that satisfies r. Thus, we see that conditions on directories can be pushed to the right until they reach the directory satisfying them. Since we cannot know which directories satisfy such conditions without evaluating them, we can no longer take find the normal form of a query before evaluating it. So, evaluation of a path would have to be done in stages. Firstly, each directory condition would be evaluated in left-to-right order, each restricted to children of directories matching previous directory conditions in the path. This will yield a set of underlying paths; the non-directory conditions would then be evaluated at each underlying path, and the results reassembled. Directories below a directory condition and above the directory satisfying that condition should contain no files, only subdirectories. When searching for directories, you are ultimately interested in the locations of such directories and the files they contain; it follows that you are not interested in files outside of matching directories.

In such an augmented system, the search for version-controlled projects under `~/source` described earlier might be represented by the directory

```
/find/alice/source/⊘(has-child "_darcs")/
```

for ⊘ the prefix for these hypothetical conditions on directories. (This example raises the need for a way to enumerate all matching files or directories in some cases, rather than traversing the restricted tree manually; this will be discussed shortly.)

Another consideration for conditions on directories is whether they should be satisfied by the deepest matching directory, the shallowest matching directory, or at a matching directory in between. All are reasonable choices; a default would need to be chosen. One might also want a way to specify a condition on files, but show the full contents of any directory containing matching files; for example, you might want to search for directories containing at least one Haskell source file, but then check what other non-Haskell files are stored in those directories. This would require a significantly more complicated query model than presented here; it would be interesting to study suitable models to find one which is both easy to use and practical to implement.

Of course, an external tool could be written to jump to the underlying directory of a filtered directory, either by modifying FindFS to expose the underlying path as an *extended attribute* of the filtered directory's virtual inode or by performing the necessary stripping of query components itself. With such a tool, you could simulate the search for directories containing Haskell source by searching for all Haskell source files, taking their parent directories, and finding the underlying paths of those directories. So some such queries could be supported with the existing model, albeit in a convoluted manner that somewhat contradicts the goal of allowing applications to make use of querying functionality unmodified.

A FindFS-aware shell

While FindFS can be used from an unmodified shell, entering conditions is significantly harder than entering normal path components, since the latter is assisted by tab-completion. Navigating a FindFS would be made easier if the shell were aware that a path component beginning with ⊘ is a condition, and suggested completions when you hit Tab: lists of possible attributes, operators, and (where reasonable) values; recently-used queries; and so on.

A common use of `find` is to run a command on every file matching the condition. A consequence of FindFS preserving the directory structure is that you cannot simply use the * shell "globbing" operator after a filtered path to enumerate all matching files: you must traverse the entire subtree. Thus it would

be useful for the shell to interpret ** as expanding to all files below that point (as is the case in certain existing shells, such as zsh). For instance, you might want to launch every Haskell source file mentioning the module System.Tracker in your text editor when its programming interface changes:

```
[/find/alice]% vim  '⊗(File:Type contains "haskell")'/⊙System.Tracker/**
```

Assume that conditions on directories have been added, as in the previous section. Now, a different operator is needed to denote the set of directories at which all directory conditions in the filtered path are satisfied; ◇, say. To see why a separate notation is needed, consider again searching for version-controlled directories within ~/source. In some cases, you might want to find all files under such directories, perhaps simply to find an approximate total of how many files you have under version control with a tool count that simply counts its arguments:

```
[/find/alice]% count  source/'⊘(has-child "_darcs")'/**
```

In other cases, you might want to pull any new upstream changes into each project directory:

```
[/find/alice]% for repo in source/'⊘(has-child "_darcs")'/◇; do
                   darcs pull --repodir=$repo
               done
```

Given conditions on directories, a slightly richer range of predicates on file names than Tracker currently allows, and the operators *, ** and ◇, we can rewrite any expression currently written as a shell glob as a FindFS path expression. So now standard glob expansions (such as foo*.txt expanding to every item in the directory whose name starts with foo, and foo.??? which matches files called foo with any three-character extension) could be removed from the shell in favour of the richer querying infrastructure provided by FindFS.

This does not defeat the original goal of exposing the functionality to unmodified applications, as we would merely be improving functionality that only the shell had in the first place. Of course, it might be useful to add similar features to the standard file chooser widgets of Gtk+, the UI toolkit used by the GNOME desktop, which would extend the same functionality to many more applications without requiring modifications to each one. In fact, that widget already supports performing simple queries with Tracker from arbitrary applications. If we were willing to only offer the features of FindFS's model to such applications, we could dispense with the virtual filesystem entirely and simply build the query model directly into the file chooser widget and the file manager. However, in reality there are many different graphical toolkits in active use, aud many useful applications which do not even use the standard file chooser window of the relevant toolkit, so there is real value in exposing the model as a filesystem.

Similar projects

Beagle is an indexing and desktop search system for the Linux desktop similar to Tracker. There is an existing Fuse-based file system, named BeagleFS, which allows you to mount individual queries and view the results as a directory. While this is ideal if you only ever want to run a handful of queries, having to mount a new file system whenever you want to run a new query is not a sensible model. The presentation of all results in a single directory also suffers from problems when multiple results share a basename, as discussed earlier.

Many modern Unix implementations have a command named locate, intended to replace the typical use of find to search for files with names matching a given pattern. locate uses a pre-built index of all file names, which is generally updated daily. This solves the repeated-effort complaiut about find for a particular class of query, but unlike FindFS does not enable richer queries.

There are a number of virtual filesystem projects sharing the name "TagFS". Some of these projects allow the user to assign arbitrary textual labels to files, producing a hierarchy of directories named for each

label containing links to each file. While this is a nice proof of concept, they do not make use of existing metadata in the files. Others represent a given class of metadata in a relevant hierarchy; for instance, one displays your music collection in a virtual *artist/album/track number - track title* hierarchy, regardless of the organisation and naming of the real files. The ability to impose such a constructed hierarchy on the results of FindFS queries would be an interesting extension.

Acknowledgements

First and foremost, thanks go to Bernard Sufrin, for the original idea and for supervising the project. Thanks also to the authors of the Fuse library and Haskell bindings, the Tracker library, and the authors of the Haskell tools and libraries used, for releasing their excellent work as free software. Finally, I would like to thank the denizens of the `#haskell` IRC channel on Freenode, for hours of assistance and interesting conversation.

Haskell source for modules of the FindFS implementation

module FindFS.Util

Types and utility functions used by FindFS. This module also re-exports several modules and some functions and types defined other modules that are needed in every module of the application.

```
{-# LANGUAGE FlexibleContexts #-}
module FindFS.Util
    ( maybeRead
    , safeHead

    , module Control.Concurrent.MVar
    , catchIOError
    , toErrno

    , lazyStopAtM

    , FindPath
    , Node (..)
    , Component (..)
    , unparsed
    , isQuery

    , Find
    , FindEnv (..)
    , FindState (..)
    , statefully

    , module Data.List
    , module Data.Maybe
    , module Data.Tuple.Utils
    , module Foreign.C.Error
    , (<$>), asks, throwError, guard, runErrorT, runReaderT, when, unless
    , get, gets, put, modify
    , MonadIO, io

    , debug
    )
where

import Control.Applicative ((<$>))

import Control.Monad
import Control.Monad.Error
import Control.Monad.Reader
import Control.Monad.Trans (MonadIO, liftIO)
import Control.Monad.State
import Control.Concurrent.MVar

import Foreign.C.Error

import System.IO.Error
import System.IO.Unsafe (unsafeInterleaveIO)

import System.Tracker (Condition, Clause)

import Data.List
import Data.Maybe
import Data.Tuple.Utils
import Data.Set (Set)
import Data.Ord
import Data.Function (on)
```

Types forming the monadic environment in which FindFS actions are executed. `statefully` wraps operations accessing the shared mutable state, serializing them relative to one another while allowing stateless operations to continue unimpeded in other threads.

```
data FindEnv = FindEnv { envCorpus :: FilePath
                       , envState :: MVar FindState
                       }

data FindState = FindState { reifiedPaths :: Set FindPath }

type Find = ReaderT FindEnv (ErrorT Errno IO)
type StatefulFind = StateT FindState Find

statefully :: StatefulFind a → Find a
statefully act = do
    stateVar ← asks envState -- Retrieve the state's box from the environment

    -- Remove the current state from the box; further attempts to read the box's
    -- contents will block until a new value is put in.
    s ← liftIO (takeMVar stateVar)

    -- Run the supplied action starting from the current state, yielding a
    -- result and a possibly-modified state.
    (result, s') ← runStateT act s

    -- Place the modified state back into the shared box, allowing threads
    -- blocking on the state to continue.
    liftIO (putMVar stateVar s')

    return result
```

Represents nodes in the virtual directory hierarchy.

```
data Node = Node { nodeName :: Component
                 , nodeVirtualPath :: FindPath
                 , nodeUnderlyingPath :: FilePath
                 , nodeTextQuery :: Maybe String
                 , nodeCondition :: Maybe Condition
                 }
    deriving (Show)
```

The representation of parsed FindFS path components. The unparsed path component is held in the representation, but is ignored for equality and ordering.

```
type FindPath = [Component]

data Component = TextSearch String FilePath -- FilePath holds unparsed path
               | Query Clause FilePath      -- component
               | NonQuery FilePath
    deriving (Show)

instance Eq Component where
    (==) = (==) `on` discardUnparsed

instance Ord Component where
    compare = comparing discardUnparsed

discardUnparsed :: Component → (Int, String)
discardUnparsed (TextSearch s _) = (1, show s)
discardUnparsed (Query      c _) = (2, show c)
discardUnparsed (NonQuery   p  ) = (3, p)

unparsed :: Component → FilePath
unparsed (TextSearch _ p) = p
unparsed (Query      _ p) = p
unparsed (NonQuery   p  ) = p

isQuery :: Component → Bool
isQuery (NonQuery _) = False
isQuery _            = True
```

A safe wrapper around `read` which does not abort the program when reading a value fails.

```
maybeRead :: Read a => String → Maybe a
```

```
maybeRead s = case reads s of
    [(x, "")] → Just x
    _         → Nothing
```

A safe replacement for `head` which does not throw an exception on an empty list.

```
safeHead :: [a] → Maybe a
safeHead (x:_) = Just x
safeHead _     = Nothing
```

Instances and functions used to work within `MonadError Errno`. `catchIOError` lifts arbritrary `IO` actions into `MonadError Errno`, catchiug any thrown unchecked exceptions and piping them into the error monad.

```
instance Error Errno where
    noMsg = eNOMSG

instance Show Errno where
    show e@(Errno i) =
        show $ errnoToIOError ("(errno " ++ show i ++ ")") e Nothing Nothing

toErrno :: IOError → Errno
toErrno ioe
    | isAlreadyExistsError ioe = eALREADY
    | isDoesNotExistError  ioe = eNOENT
    | isAlreadyInUseError  ioe = eBUSY
    | isFullError          ioe = eAGAIN
    | isEOFError           ioe = eIO
    | isIllegalOperation   ioe = eNOTTY
    | isPermissionError    ioe = ePERM
    | otherwise                = eFAULT


-- The FlexibleContexts language extension is needed for this type signature.
catchIOError :: (MonadError Errno eio, MonadIO eio) => IO a → eio a
catchIOError act = do
    either ← io $ try act
    case either of
        (Left e)  → throwError $ toErrno e
        (Right r) → return r

io :: (MonadIO m) => IO a → m a
io = liftIO
```

`lazyStopAtM` runs the supplied actions in order as the result list is evaluated, stopping at and including the first time the predicate is `True`. It is used to represent the full result set of a query as a lazy list.

```
stopAtM :: Monad m => (a → Bool) → [m a] → m [a]
stopAtM = wrappedStopAtM id

lazyStopAtM :: (a → Bool) → [IO a] → IO [a]
lazyStopAtM = wrappedStopAtM unsafeInterleaveIO

wrappedStopAtM wrapper p = stopAtM'
    where stopAtM' []     = return []
          stopAtM' (m:ms) = do res ← m
                               if p res
                                then return [res]
                                else do ress ← wrapper $ stopAtM' ms
                                        return (res:ress)
```

A wrapper providing a single place to disable debugging output if necessary.

```
debug :: MonadIO io
      => String
      → io ()
debug = io . putStrLn
```

module FindFS.Path

This module exports a single function, `parsePath`, which breaks a FindFS path into a `Node` specifying its underlying path and its filter. The query components are parsed using the Parsec parser combinator library, which allows the query syntax to be expressed directly in a monadic style.

```
module FindFS.Path
    (parsePath)
where

import System.FilePath

import System.Tracker.Query
import qualified  System.Tracker.Query.Properties as Pr

import Text.ParserCombinators.Parsec

import FindFS.Util
```

`parsePath'` accepts two paths as arguments: the first is a path relative to the root of the FindFS, and the second is the root of the underlying corpus. In effect, it normalizes the path, then transforms the query components into the data types required by the Tracker client library.

`parsePath` is a very thin wrapper around it in `Find` monad, taking the root of the corpus from the **`Reader`** environment and throwing an appropriate **`Errno`** when the path cannot be parsed.

```
parsePath :: FilePath → Find Node
parsePath virtualPath = do
    prefix ← asks envCorpus
    case parsePath' virtualPath prefix of
        Just n  → return n
        Nothing → throwError eNOENT


parsePath' :: FilePath
           → FilePath
           → Maybe Node
parsePath' virtualPath prefix = do
    components ← runP path virtualPath
    let (nonQueries, clauses, textQueries) = splitComponents components

    let underlying = prefix </> joinPath nonQueries

    let condition = case clauses of
            []  → Nothing
            [c] → Just (Condition c)
            cs  → Just (Condition (And cs))

    let textQuery = case textQueries of
            [] → Nothing
            _  → Just $ unwords textQueries

    return $ Node { nodeName = last components
                  , nodeVirtualPath = components
                  , nodeUnderlyingPath = underlying
                  , nodeTextQuery = textQuery
                  , nodeCondition = condition
                  }
```

This function runs a Parsec parser, discarding the error message on parse failure; there is no way the message could be passed back to the FindFS user.

```
runP :: Parser a → String → Maybe a
runP p input = case runParser p () "" input of
                 Left err → Nothing
                 Right x  → Just x
```

A combinator to run a parser, then return its result along with the string of input it consumed.

```
consumed :: Parser a → Parser (a, String)
consumed p = do input ← getInput

                start ← sourceColumn <$> getPosition
                result ← p
                end ← sourceColumn <$> getPosition

                let str = take (end - start) input

                return (result, str)
```

A path is parsed into a list of components by applying each component parser in turn after each slash; that list of components can then be divided into lists of each kind of component, effectively normalizing the path.

```
splitComponents :: [Component] → ([FilePath], [Clause], [String])
splitComponents [] = ([], [], [])
splitComponents (c:cs) = case c of TextSearch s  _ → (  ps,    cls, s:ss)
                                   Query      cl _ → (  ps, cl:cls,   ss)
                                   NonQuery   p    → (p:ps,    cls,   ss)
    where (ps, cls, ss) = splitComponents cs


path :: Parser [Component]
path =  end
    <|> (char '/' >> (end <|> more))
  where end = eof >> return []
        more = do component ← choice [ try nonQueryComponent
                                     , try clauseComponent
                                     , textSearchComponent
                                     ]
                  components ← path
                  return (component:components)

nonQueryComponent :: Parser Component
nonQueryComponent = do cs ← many1 (noneOf "/")
                       guard . not $ any (`isPrefixOf` cs) [odot, otimes]
                       return $ NonQuery cs
```

We define a parser for a ⊗ path component to match a string starting with that symbol, followed by a clause which is returned (along with the consumed portion of the input). A clause is either an atomic predicate, or the conjunction or disjunction of two other clauses, in either case wrapped in parentheses.

```
clauseComponent :: Parser Component
clauseComponent = do (c, str) ← consumed (string otimes >> clause)
                     return $ Query c str

clause :: Parser Clause
clause = do char '('
            c ← atom <|> combinedClauses
            char ')'
            return c

combinedClauses :: Parser Clause
combinedClauses = do c1 ← clause
                     skipMany space
                     op ← choice [ string "and" >> return And
                                 , string "or"  >> return Or
                                 ]
                     skipMany space
                     c2 ← clause
                     return (op [c1, c2])
```

Tracker's property names take the form *type of file:name of field*, such as `Image:CameraMake` and `Audio:TrackNo`. The property corresponding to the type of a file is named `File:Mime`, and contains the file's MIME type, such as `audio/mpeg`. Unfortunately, all MIME types contain a forward slash, which cannot appear in path components as it is the directory separator! To work around this, FindFS

recognises the additional property `File:Type`, and treats (`File:Type == "foo"`) as if you had written (`File:Mime contains "foo"`). Other property names are checked against `Pr.names`, a list of valid property names supplied by Tracker.

```
atom :: Parser Clause
atom = do p ← property
          skipMany1 space
          o ← operator
          skipMany1 space
          v ← value
          if p == prop_type
            then
                return $ Atom (if o == Equals then Contains else o) prop_mime v
            else
                return $ Atom o p v
    where prop_type = Property "File:Type"
          prop_mime = Property "File:Mime"

property :: Parser Property
property = do name ← many1 (letter <|> char ':')
              let p = Property name
              guard (name == "File:Type" || p `elem` Pr.names)
              return $ Property name
```

Valid operators are matched by an entry in the lookup table, and mapped to the corresponding value of type `Operator`. (Matching elements of a set has not been implemented in FindFS, although it is supported by Tracker.)

```
operator :: Parser Operator
operator = choice $ map p operators
    where operators = [("==",          Equals)
                      ,(">=",          GreaterEqual)
                      ,(">",           GreaterThan)
                      ,("<=",          LessEqual)
                      ,("<",           LessThan)
                      ,("contains",    Contains)
                      ,("matches",     Regex)
                      ,("startswith",  StartsWith)
                    --,("in",          InSet)
                      ]
          p (symbol, result) = try (string symbol >> return result)
```

A value in an atomic clause can be an integer (with optional unit specification), a double-quoted string, or a date. Whether the units match the property is not checked.

```
value :: Parser Value
value = choice $ map try [dateValue, intValue, stringValue]

intValue :: Parser Value
intValue = do ds ← many1 digit
              let (Just i) = maybeRead ds
              s ← option 1 scale
              return (IntV (i * s))
    where unit  = skipMany space >> (choice $ map (string . fst) units)
          units = [("kb", 1024), ("mb", 1024 ^ 2), ("gb", 1024 ^ 3), ("b", 1)]
          scale = do u ← unit
                     return . fromJust $ lookup u units

stringValue :: Parser Value
stringValue = do char '"'
                 s ← many1 letter
                 char '"'
                 return (StringV s)
```

Ideally, entering a date as "4 years ago" would be supported, but currently only YYYY-MM-DD HH:MM:SS is accepted.

```
dateValue :: Pareer Value
dateValue = do year ← count 4 digit
```

```
        char '-'
        month ← count 2 digit
        char '-'
        day ← count 2 digit
        char ' '
        hour ← count 2 digit
        char ':'
        min ← count 2 digit
        char ':'
        sec ← count 2 digit

        return . DateV . intercalate " " $ [intercalate "-" [year, month, day]
                                           ,intercalate ":" [hour, min, sec]
                                           ]
```

Parsing ⊙ path components is straightforward: match ⊙ plus a string of at least one character, and return that string (along with the string of input consumed).

```
textSearchComponent :: Parser Component
textSearchComponent = do (terms, str) ← consumed (string odot >> many1 (noneOf "/"))
                         return $ TextSearch terms str
```

There are currently some encoding issues when embedding Unicode entities in literal strings. To work around this, we define variables corresponding to the three bytes making up the UTF-8 encodings of ⊗ and ⊙.

```
otimes = "\226\138\151"
odot   = "\226\138\153"
```

module FindFS.Node

This module contains functions to inspect nodes of the virtual directory hierarchy, querying Tracker if necessary. (The type `Node` is defined in FindFS.Util to avoid a circular dependency between this module and FindFS.Path.)

```
{-# LANGUAGE FlexibleContexts, RecordPuns #-}
module FindFS.Node
    ( evaluateDir
    , evaluateAtPath
    , evaluateAtUnfilteredPath

    , isFiltered

    , failOnFiltered
    , failOnQueryLeaf
    )
where

import qualified System.Directory as D
import System.FilePath
import System.Tracker

import FindFS.Util
import FindFS.Path (parsePath)
```

A convenience function determining whether a node is filtered.

```
isFiltered :: Node → Bool
isFiltered n = isJust (nodeTextQuery n) || isJust (nodeCondition n)
```

Actions to throw an "access denied" exception on filtered nodes, and on nodes whose last component is a query, respectively.

```
failOnFiltered :: Node → Find ()
failOnFiltered node = when (isFiltered node) (throwError eACCES)
```

```
failOnQueryLeaf :: Node → Find ()
failOnQueryLeaf node@(Node {nodeName }) =
    when (isQuery nodeName) $
        throwError eACCES
```

Gets the contents of the directory at the supplied node. If it is filtered, then the results are fetched from Tracker, and restricted to children of the supplied node; otherwise, `getDirectoryContents` is called on the underlying path.

```
evaluateDir :: Node
            → Find [FilePath]
evaluateDir n@(Node { nodeUnderlyingPath })
    | isFiltered n = do
        exists ← io $ D.doesDirectoryExist nodeUnderlyingPath
        when (not exists) $ throwError eNOENT

        results ← searchNode n

        let relevant = takeRelevant results nodeUnderlyingPath
        let contents = ".":"..":firstLevelObjects relevant
        return contents
    | otherwise  = do
        catchIOError $ D.getDirectoryContents nodeUnderlyingPath

firstLevelObjects :: [FilePath] → [FilePath]
firstLevelObjects paths =
    nub $ map (head . splitDirectories) paths

takeRelevant :: [FilePath] → FilePath → [FilePath]
takeRelevant fullpaths target = catMaybes $ map (stripPrefix target') fullpaths
    where target' = addSlash target
          addSlash p = if last p == '/' then p else p ++ "/"
```

If given a query-free path, the supplied action is run on it; if the path is filtered, an "access denied" exception is thrown.

```
evaluateAtUnfilteredPath :: (FilePath → IO a)
                         → FilePath
                         → Find a
evaluateAtUnfilteredPath act path = do
    node ← parsePath path
    failOnFiltered node
    catchIOError $ act (nodeUnderlyingPath node)
```

Given an `IO` action and a virtual path, the path is parsed and the action run on the underlying path of the resulting node.

```
evaluateAtPath :: (FilePath → IO a)
               → FilePath
               → Find a
evaluateAtPath act path = do
    node ← parsePath path
    (act `evaluateAt` node)
```

Given an unfiltered node, the supplied action is simply run on the underlying path.

If the supplied node is filtered, the set of paths matching its filter is found, and we check that the node's underlying path is in that list before applying the action. If not, a "file not found" exception is thrown.

```
evaluateAt :: (FilePath → IO a)
           → Node
           → Find a
evaluateAt act node@(Node {nodeUnderlyingPath}) = do
    when (isFiltered node) $ do
        let pathElts = splitDirectories nodeUnderlyingPath

        matches ← searchNode node

        -- break the matches into a list of path components
```

```
        let matchEltss = map splitDirectories matches

        unless (any (pathElts `isPrefixOf`) matchEltss) $
            throwError eNOENT
    catchIOError $ act nodeUnderlyingPath
```

`searchAll` retrieves all results matching the supplied query terms and conditions from the Tracker daemon. The results are retrieved lazily, in batches of `chunkSize`.

Since the list of matches may change in between calls to `searchQuery` (if the Tracker daemon indexes new files in another thread, for instance), we must use `nub` to remove duplicates from the list of results. This is made slightly more likely by the use of `unsafeInterleaveIO` in `lazyStopAtM`, but would be possible anyway.

`searchNode` is a wrapper pulling the queries out of the node and passing them to `searchAll`. It does **not** restrict the results to those relevant to the node; this is done by the actions using `searchNode`.

```
searchAll  :: MonadIO io
           => Maybe String
           → Maybe Condition
           → io [FilePath]
searchAll terms cond = io . withTrackerErr $ \client →
    let chunkSize = 100 -- If this is too large, the D-Bus method times out
                        -- because Tracker takes too long
        fetchChunk i = do
            debug $ "Retrieving " ++ show (i*chunkSize) ++ "..."
            searchQuery Files [] terms Nothing cond (i * chunkSize) chunkSize False client
        fetchers = map fetchChunk [0..]

        noMoreResults xs = length xs < chunkSize

    in do results ← lazyStopAtM noMoreResults fetchers
          return $ (nub . map fst3 . concat) results

searchNode :: Node → Find [FilePath]
searchNode node@(Node { nodeTextQuery, nodeCondition }) = do
    searchAll nodeTextQuery nodeCondition
```

module FindFS

This is the main module of FindFS, which implements the Fuse callbacks.

```
{-# LANGUAGE RecordPuns #-}
module Main where

import Control.Exception hiding (catch)
import System.Directory (getHomeDirectory, doesDirectoryExist, getDirectoryContents)
import System.FilePath (takeFileName, (</>), splitDirectories)
import System.IO
import System.IO.Error
import System.Posix.Directory
import System.Posix.Files
import System.Posix.IO
import System.Posix.Types

import Data.Bits ((.&.))
import Data.Function (on)

import System.Environment
import System.Exit (exitFailure)

import qualified Data.ByteString as B

import qualified Data.Set as Set

import System.Fuse
import qualified System.Tracker as T
```

```
import FindFS.Util
import FindFS.Node
import FindFS.Path
```

Virtual file descriptors are represented by the open file descriptor for the underlying file.

```
type FindFileHandle = Fd
```

The main action of the application determines the user's home directory. From this, it determines the correct mountpoint, constructs the environment and then hands a set of callbacks to Fuse. It also passes an exception handler which is used as a last resort when a callback throws an unchecked exception.

```
main :: IO ()
main = do
    corpus ← getHomeDirectory

    state ← newMVar $ FindState { reifiedPaths = Set.empty }
    let env = FindEnv { envState = state
                      , envCorpus  = corpus
                      }
    let mountpoint = "/find" </> (takeFileName corpus)
    args ← getArgs
    withArgs (args ++ [mountpoint]) $ do
        fuseMain (findFSOps env) findExceptionHandler

findExceptionHandler :: Exception → IO Errno
findExceptionHandler e = do
    debug $ "exception reached findExceptionHandler: " ++ show e
    case e of (IOException ioe) → return $ toErrno ioe
              _                 → return eFAULT
```

All of the implementations of Fuse operations here are in the `Find` monad. They are supplied to Fuse as `IO` actions by wrapping them in a function `run` which reconstitutes the necessary monad from `env`. `void` is used to allow actions which would otherwise have to return `eOK` on success to return `()`, as is normal for successful actions.

```
findFSOps :: FindEnv → FuseOperations FindFileHandle
findFSOps env =
    defaultFuseOps
        { fuseGetFileStat         = run . findGetFileStat
        , fuseReadSymbolicLink    = run . findReadSymbolicLink
        , fuseCreateDevice        = \p t m i → runVoid $ findCreateDevice p t m i
        , fuseCreateDirectory     = \p m → runVoid $ findCreateDirectory p m
        , fuseRemoveLink          = runVoid . findRemoveLink
        , fuseRemoveDirectory     = runVoid . findRemoveDirectory
        , fuseCreateSymbolicLink = \src dst → runVoid $ findCreateSymbolicLink src dst
        , fuseRename              = \src dst → runVoid $ findRename src dst
        , fuseCreateLink          = \src dst → runVoid $ findCreateLink src dst

        , fuseSetFileMode         = \p m → runVoid $ findSetFileMode p m
        , fuseSetOwnerAndGroup    = \path u g → runVoid $ findSetOwnerAndGroup path u g
        , fuseSetFileSize         = \path s → runVoid $ findSetFileSize path s
        , fuseSetFileTimes        = \path a m → runVoid $ findSetFileTimes path a m
        , fuseSynchronizeFile     = \path sync → runVoid $ findSynchronizeFile path sync

        , fuseOpen                = \path mode flags → run $ findOpen path mode flags
        , fuseRead                = \path h count off → run $ findRead path h count off
        , fuseWrite               = \path h buf off → run $ findWrite path h buf off
        , fuseFlush               = \path h → runVoid $ findFlush path h
        , fuseRelease             = \path h → run (findRelease path h) >> return ()

        , fuseGetFileSystemStats = run . findGetFileSystemStats

        , fuseOpenDirectory       = runVoid . findOpenDirectory
        , fuseReadDirectory       = run . findReadDirectory

        , fuseInit                = return ()
        }
    where run :: Find a → IO (Either Errno a)
```

```
run action = runErrorT $ runReaderT action env

void :: Either Errno () → Errno
void = either id (const eOK)

runVoid :: Find () → IO Errno
runVoid act = void <$> run act
```

Symbolic links are permitted to have relative paths as their targets. This poses a problem if the relative path uses the .. directory entry to refer to its parent directory, as the parent of a filtered directory is not necessarily the parent of the underlying directory. Suppose you have a script `/src/utils/foo`, and a symbolic link `/bin/foo` pointing to `../src/utils/foo`. An application evaluating the link relative to `/find/user/bin` will reach the correct target, but following the link from `/find/user/bin/⊙python` would incorrectly lead to `/find/user/bin/src/utils/foo`.

We could attempt to introduce extra `../` elements to compensate for any such inconsistency, but for simplicity we simply forbid reading such links at a filtered path.

```
findReadSymbolicLink :: FilePath → Find FilePath
findReadSymbolicLink path = do
    n@Node { nodeUnderlyingPath = p' } ← parsePath path
    target ← io $ readSymbolicLink p'
    let components = splitDirectories target
    when (isFiltered n && ".." `elem` components) $ throwError eACCES
    return target
```

The intention was to attach the reification of query paths to `mkdir` and `rmdir`. However, this proved to be impossible: the Fuse library calls `findGetFileStat` on any path passed to `mkdir` to check if it already exists before calling `findCreateDirectory`, which of course they do since query paths exist when examined directly. This could probably be fixed by modifying the Fuse library, although it might be a limitation of the kernel. Instead, reification is performed and reversed by setting or clearing the "sticky" permission bit on a query directory (from the shell, this is done with the `chmod +t` command).

```
findSetFileMode :: FilePath → FileMode → Find ()
findSetFileMode path mode = do
    node@(Node { nodeName, nodeUnderlyingPath }) ← parsePath path
    if isQuery nodeName
        then setReified node (hasStickyBit mode)
        else do failOnFiltered node
                io $ setFileMode nodeUnderlyingPath mode

hasStickyBit :: FileMode → Bool
hasStickyBit mode = mode .&. stickyBit /= 0
    where stickyBit = fromIntegral 0o1000

setReified :: Node → Bool → Find ()
setReified n@(Node { nodeVirtualPath }) reify = statefully $ do
    paths ← gets reifiedPaths
    let update = if reify then Set.insert else Set.delete
    let newPaths = update nodeVirtualPath paths
    modify (\st → st { reifiedPaths = newPaths })
```

Then, when reading the contents of a directory we add any reified query components which should appear. They are given the same `FileStat` as the directory being read, as they are filtered views of it.

When listing a filtered directory, Tracker may return matches for files that have been deleted since it last indexed that directory. So, while statting each real entry in the directory, we discard any entries that turn out not to actually exist.

```
findReadDirectory :: FilePath → Find [(FilePath, FileStat)]
findReadDirectory path = do
    node@(Node { nodeUnderlyingPath }) ← parsePath path
    contents ← evaluateDir node
    withStats ← io $ catMaybes <$> mapM (getDetails nodeUnderlyingPath) contents

    reifiedContents ← getReifiedFor node
    selfStat ← fileStatusToFileStat <$> io (getSymbolicLinkStatus nodeUnderlyingPath)
```

```
    let reifiedWithStat = zip reifiedContents (repeat selfStat)

    return $ withStats ++ reifiedWithStat

getDetails :: FilePath → FilePath → IO (Maybe (FilePath, FileStat))
getDetails root name =
    do status ← getSymbolicLinkStatus (root </> name)
       return $ Just (name, fileStatusToFileStat status)
    `catch` \e → if isDoesNotExistError e
                 then return Nothing
                 else ioError e
```

The list of reified directories that need to be added is found by looking through the list of all reified directories for those starting with the current directory, followed by a query component. For example, if both `recipes/⊙cake` and `recipes/hypothetical/⊙cake` have both been reified, and we are listing the contents of `recipes`, we need only add `⊙cake` to the results; `hypothetical` is an ordinary directory, so will already be present.

```
getReifiedFor :: Node → Find [FilePath]
getReifiedFor (Node { nodeVirtualPath }) = do
    allReified ← statefully $ Set.elems <$> gets reifiedPaths
    let relevant = [ c
                   | Just c ← map (componentAfter nodeVirtualPath) allReified
                   , isQuery c
                   ]
    return $ map unparsed relevant

componentAfter :: FindPath → FindPath → Maybe Component
componentAfter p q = do
    suffix ← stripPrefix p q
    safeHead suffix
```

Before moving an nnderlying file, we check that the destination is unfiltered, and that if the source is filtered its last component is a non-qnery. (This disallows `mv /find/alice/bar/⊙foo /find/alice`, which would otherwise try to move a query, but permits `mv /find/alice/⊙foo/bar /find/alice`.)

```
findRename :: FilePath → FilePath → Find ()
findRename src dest = do
    srcNode ← parsePath src
    destNode ← parsePath dest

    failOnFiltered destNode
    failOnQueryLeaf srcNode

    io $ (rename `on` nodeUnderlyingPath) srcNode destNode
```

Removing files is permitted, even if the path to them is filtered, as previously discussed.

```
findRemoveLink :: FilePath → Find ()
findRemoveLink = evaluateAtPath removeLink
```

When removing a filtered directory not ending in a query component, we check if the underlying directory is empty; if so, it is removed as well. Unfiltered directories are removed as normal. Removing a filtered directory ending in a query component was intended to unreify it, but as mentioned reification is performed using the sticky bit instead.

```
findRemoveDirectory :: FilePath → Find ()
findRemoveDirectory path = do
    node@(Node {nodeUnderlyingPath}) ← parsePath path
    if isFiltered node
        then do failOnQueryLeaf node
                underlyingContents ← io $ getDirectoryContents nodeUnderlyingPath
                when (sort underlyingContents == [".", ".."])
                    (io $ removeDirectory nodeUnderlyingPath)
        else io $ removeDirectory nodeUnderlyingPath
```

When a proxy file is opened, we open a handle on the underlying file. This handle is passed to subsequent operations on the virtual filehandle by Fuse, and operations are delegated to it. When the proxy is closed, so is the underlying handle.

```
findOpen :: FilePath → OpenMode → OpenFileFlags
         → Find FindFileHandle
findOpen path mode flags = do
    let open p = openFd p mode Nothing flags
    fd ← (open `evaluateAtPath` path)
    return fd

findRead :: FilePath → FindFileHandle → ByteCount → FileOffset
         → Find B.ByteString
findRead path fd count off = do
    newOff ← io $ fdSeek fd AbsoluteSeek off
    if off /= newOff
        then throwError eINVAL
        else io $ do handle ← fdToHandle fd
                     B.hGetNonBlocking handle (fromIntegral count)

findWrite :: FilePath → FindFileHandle → B.ByteString → FileOffset
          → Find ByteCount
findWrite path fd buf off = do
    newOff ← io $ fdSeek fd AbsoluteSeek off
    if off /= newOff
        then throwError eINVAL
        else io $ do handle ← fdToHandle fd
                     B.hPut handle buf
                     return (fromIntegral $ B.length buf)

findRelease :: FilePath → FindFileHandle → Find ()
findRelease _ fd = io $ closeFd fd
```

All other operations are simply delegated to the underlying file, in some cases failing if the snpplied path is filtered.

```
fileStatusToEntryType :: FileStatus → EntryType
fileStatusToEntryType status
    | isSymbolicLink     status = SymbolicLink
    | isNamedPipe        status = NamedPipe
    | isCharacterDevice status = CharacterSpecial
    | isDirectory        status = Directory
    | isBlockDevice      status = BlockSpecial
    | isRegularFile      status = RegularFile
    | isSocket           status = Socket
    | otherwise                 = Unknown

fileStatusToFileStat :: FileStatus → FileStat
fileStatusToFileStat status =
    FileStat { statEntryType        = fileStatusToEntryType status
             , statFileMode         = fileMode status
             , statLinkCount        = linkCount status
             , statFileOwner        = fileOwner status
             , statFileGroup        = fileGroup status
             , statSpecialDeviceID  = specialDeviceID status
             , statFileSize         = fileSize status
             -- fixme: 1024 is not always the size of a block
             , statBlocks           = fromIntegral (fileSize status `div` 1024)
             , statAccessTime       = accessTime status
             , statModificationTime = modificationTime status
             , statStatusChangeTime = statusChangeTime status
             }

findGetFileStat :: FilePath → Find FileStat
findGetFileStat path = do
    status ← getSymbolicLinkStatus `evaluateAtPath` path
    return $ fileStatusToFileStat status

findOpenDirectory :: FilePath → Find ()
findOpenDirectory path = do
```

```
    exists ← doesDirectoryExist `evaluateAtPath` path
    unless exists $ throwError eNOENT

findCreateDevice :: FilePath → EntryType → FileMode → DeviceID → Find ()
findCreateDevice path entryType mode dev = do
    let combinedMode = entryTypeToFileMode entryType `unionFileModes` mode
        create p = createDevice p combinedMode dev
    (create `evaluateAtUnfilteredPath` path)

findCreateDirectory :: FilePath → FileMode → Find ()
findCreateDirectory path mode = do
    let create p = createDirectory p mode
    (create `evaluateAtUnfilteredPath` path)

findCreateSymbolicLink :: FilePath → FilePath → Find ()
findCreateSymbolicLink src dest = do
    let mkLink p = createSymbolicLink p dest
    mkLink `evaluateAtUnfilteredPath` src

findCreateLink :: FilePath → FilePath → Find ()
findCreateLink src dest = do
    srcNode ← parsePath src
    destNode ← parsePath dest
    mapM_ failOnFiltered [srcNode, destNode]
    io $ (createLink `on` nodeUnderlyingPath) srcNode destNode

findSetOwnerAndGroup :: FilePath → UserID → GroupID → Find ()
findSetOwnerAndGroup path uid gid = do
    let set p = setOwnerAndGroup p uid gid
    (set `evaluateAtUnfilteredPath` path)

findSetFileSize :: FilePath → FileOffset → Find ()
findSetFileSize path off = do
    let set p = setFileSize p off
    (set `evaluateAtUnfilteredPath` path)

findSetFileTimes :: FilePath → EpochTime → EpochTime → Find ()
findSetFileTimes path accessTime modificationTime = do
    let set p = setFileTimes p accessTime modificationTime
    (set `evaluateAtUnfilteredPath` path)

findGetFileSystemStats :: FilePath → Find FileSystemStats
findGetFileSystemStats _ = throwError eOK

findFlush :: FilePath → FindFileHandle → Find ()
findFlush _ fd = return ()

findSynchronizeFile :: FilePath → SyncType → Find ()
findSynchronizeFile _ _ = return ()
```