Forest: A Language and Toolkit for Programming with Filestores

Kathleen Fisher
Tufts University

Nate Foster
Cornell University

David Walker
Princeton University

Kenny Q. Zhu
Shanghai Jiao Tong University

Abstract

A *filestore* is a structured collection of data files housed in a conventional hierarchical file system. Many applications use filestores as a poor-man's database, and the correct execution of these applications requires that the collection of files, directories, and symbolic links stored on disk satisfy a variety of precise invariants. Moreover, all of these structures must have acceptable ownership, permission, and timestamp attributes. Unfortunately, current programming languages do not provide support for documenting assumptions about filestores, detecting errors in them, or safely loading from and storing to them.

This paper describes the design, implementation, and semantics of Forest, a new domain-specific language for describing filestores. The language uses a type-based metaphor to specify the expected structure, attributes, and invariants of filestores. Forest generates loading and storing functions that make it easy to connect data on disk to an isomorphic representation in memory that can be manipulated as if it were any other data structure. Forest also generates metadata that describes the degree to which the structures on the disk conform to the specification, making error detection easy. In a nutshell, Forest extends the rigorous discipline of typed programming languages to the untyped world of file systems.

We have implemented Forest as an embedded domain-specific language in Haskell. In addition to generating infrastructure for reading, writing and checking file systems, our implementation generates type class instances that make it easy to build generic tools that operate over arbitrary filestores. We illustrate the utility of this infrastructure by building a file system visualizer, a file access checker, a generic query interface, description-directed variants of several standard UNIX shell tools and (circularly) a simple Forest description inference engine. Finally, we formalize a core fragment of Forest in a semantics inspired by classical tree logics and prove round-tripping laws showing that the loading and storing functions behave sensibly.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages

General Terms Languages, Design, Theory

Keywords Data description languages, file systems, filestores, domain-specific languages, ad hoc data, Haskell, bidirectional transformations, generic programming

1. Introduction

Databases are an effective, time-tested technology for storing structured and semi-structured data. Nevertheless, many computer users eschew the benefits of structured databases and store important semi-structured information in collections of files and directories in a conventional file system instead. For example, the Princeton Computer Science Department stores records of undergraduate student grades in a structured set of directories and uses scripts to compute averages and study grading trends. Similarly, Michael Freedman collects sets of log files from CoralCDN, a distributed content distribution network [11, 12]. The logs are organized in hierarchical directory structures based on machine name, time and date. Freedman mines the logs for information on system security and performance. At Harvard, physics professor Vinothan Manoharan stores his experimental data in sets of files and extracts information using python scripts. At AT&T, vast structured repositories contain network monitoring information, phone call records, and billing data. Many software code bases, including Haskell and its associated Cabal libraries, require that specific files exist in particular formats at precise locations described in other files. Similarly, version control systems like `cvs` utilize the file system to store revision information. Web sites require various types of files to exist in particular directories according to content type, and security considerations often require particular permissions on these files. Many other examples exist across the computational sciences and social sciences, in computer systems research, in computer systems administration and in industry.

Users choose to implement ad hoc databases in this manner for a number of reasons. A key factor is that using databases often requires paying substantial up-front costs such as: (1) finding and evaluating the appropriate database software (and possibly paying for it); (2) learning how to load data into the database; (3) writing programs to transform the raw data for loading into the database; (4) learning how to access the data once it is in the database; and (5) interfacing the database with a conventional programming language to support applications that use the data. Finally, it may be the case that the database optimizes for a pattern of use not suited to the actual application, which makes the overhead of the database system even less desirable.

Rather than paying these costs, programmers often store data in the file system, using a combination of directory structure, file names, file contents, and symbolic links to organize the data. We call such a data representation a *filestore*. The "query language" for a filestore is often a shell script or conventional programming language.

Unfortunately, despite their initial convenience, using filestores can have a number of negative consequences. First, there is generally no documentation, which means it can be hard to understand the data and its organization. New users struggle to learn the structure, and if the system administrator leaves, knowledge of the data organization may be lost. Second, the structure of the filestore tends to evolve: new elements are added and old formats are changed,

ICFP'11, September 19–21, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0865-6/11/09. . . $10.00

sometimes accidentally. Such evolution can cause hacked-up data processing tools to break or return erroneous results; it also further complicates understanding the data. Third, there is often no systematic means for detecting errors even though data errors can be immensely important. For example, for filestores containing monitoring information, errors can signal that some portion of the monitored system is broken. Fourth, analyses tend to be built from scratch. There is no auxiliary query or tool support and no help with debugging. Tools tend to be "one-off" efforts that are not reuseable. Fifth, dealing with large data sets, which are common in this setting, imposes extra difficulties. For example, standard shell tools such as `ls` fail when more than 256 files appear on the command line. Hence, programmers must break up their data and process it in smaller sets, a tedious task.

We propose a better way: A type-based specification language, programming environment and toolkit for describing and managing filestores. This language, called Forest, is implemented as an embedded domain-specific language in Haskell. Forest allows programmers to describe the expected shape of a filestore and to materialize it as typed, format-specific Haskell data structures. Conversely, given data structures with the appropriate type, Forest makes it straightforward to dematerialize these structures and write them out to disk.

The first benefit of the Forest system is that Forest descriptions provide *executable documentation* that can be used to check whether a given filestore conforms to its specification. For example, Unix file systems should be laid out according to the informal standard set forth by the Filesystem Hierarchy Standard Group [3], which requires, among other things, that certain directories *only* contain certain files, presumably for security reasons. Forest provides a language for expressing standards precisely and for checking that given file systems conform to them. As another example, the Pads website [25] contains a complex set of scripts and data files to implement an online demo. Unless all of the required data files, directories, and symbolic links are configured correctly, the web demo fails with an inscrutable error message. Forest allows the Pads webmaster to precisely document all of these requirements and to detect specification violations, making it easy to find and repair errors. And, of course, if the current webmaster were to leave her post, her successor could use the Forest description to help understand the system.

As well as serving as executable documentation, Forest provides substantial additional support for programmers. The goal is for programmers to obtain a whole range benefits by writing one simple, compact file system specification. The automatically generated auxiliary support includes: (1) a set of type declarations to represent the filestore in memory; (2) a set of type declarations that capture errors and file system attributes for the filestore; (3) a loading function to populate these in-memory structures; (4) a storing function to push possibly updated structures back out to disk; (5) type class instance declarations that make it possible for programmers to query, analyze, and transform filestore data using generic functions; and (6) a set of useful generic functions/scripts that operate over instances of these type classes.

The main contribution of this work is conceptual: We propose the idea of extending a modern programming language with tightly integrated linguistic features for describing and manipulating filestores. To demonstrate the potential of this idea, the following sections flesh out our proposal in greater depth:

- Section 2 begins with two concrete motivating examples, drawn from the authors' day-to-day experience managing computer systems. While there are just two central examples in this paper, the Forest web site [9] contains a number of further examples and case studies.
- Section 3 describes a concrete language design. The design is characterized by a simple, intuitive and compositional syntax that is tightly integrated with Haskell, our host language. The design is also tightly integrated with Pads/Haskell, a domain-specific language for describing individual files (as opposed to entire filestores), inspired by past work on related data description languages [5, 6, 7, 22]. This tight integration was a crucial design goal as it allows programmers to transition seamlessly between ordinary Haskell data structures, file internals and file collections, all in a uniform syntax.
- Section 4 explains how to write Haskell programs that operate over filestores described in Forest. The goal of this section is to provide a sense of how easy it is to write simple filestore scripts or queries.
- Section 5 shows that it is possible to use Forest to make the management of filestores even easier by developing generic tools capable of operating over any filestore. We have developed several such tools including a generic query interface, a file system visualization tool, an access-control permission checker, and a series of UNIX-like scripting tools. We have also built a simple description-inference tool to help users write a new description for an existing file system. These tools are interesting in their own right and also as case studies of putting generic programming techniques into practice. In addition, they provide evidence that our design is effectively integrated into the Haskell ecosystem.
- Section 6 explains our implementation, which is complete and may be downloaded at the Forest web site [9]. In addition to delivering a useful tool, our engineering work has the auxiliary benefit of serving as a case study in domain-specific language implementation. In fact, it has already had significant impact as such: the Haskell team modified and extended Haskell's quasiquoting mechanism in response to our needs.
- Section 7 describes the formal semantics for core Forest and states theorems demonstrating that the mappings between the filestore and in-memory structures behave correctly. These theorems are inspired by the "round-tripping" laws for well-behaved lenses [10], but are significantly more complicated as the load and store functions have to deal with inconsistencies stemming from dependencies, duplication, and invalid data.
- Section 8 contains a discussion of related work. There has been much past work on domain-specific languages for describing, parsing and printing individual data files. Examples include Lex, Yacc, Antlr [26], Parsec [19] and Pads [7], to name just a few. However, Forest differs substantially from any of these systems because it focuses on technology for describing *entire filestores*. A key difference is that simple filestores are *trees* and complex ones with symbolic links are *graphs*, whereas files are *sequences* (of characters or tokens). Consequently, the language design, formal systems, semantic issues, and underlying implementation technology are all entirely different.

2. Example Filestores

In this section, we present two example filestores. We use these examples to motivate and explain the design of Forest.

The first filestore contains information about students in Princeton's undergraduate computer science program. The faculty use the information to decide on undergraduate awards and to track grading trends. Its format has changed over time—something that is typical for ad hoc filestores. Naturally, any description needs to cope with the variations introduced as formats evolve.

Figure 1 shows a snippet of the (anonymized) student filestore designed to illustrate its structure. At the top level, there are

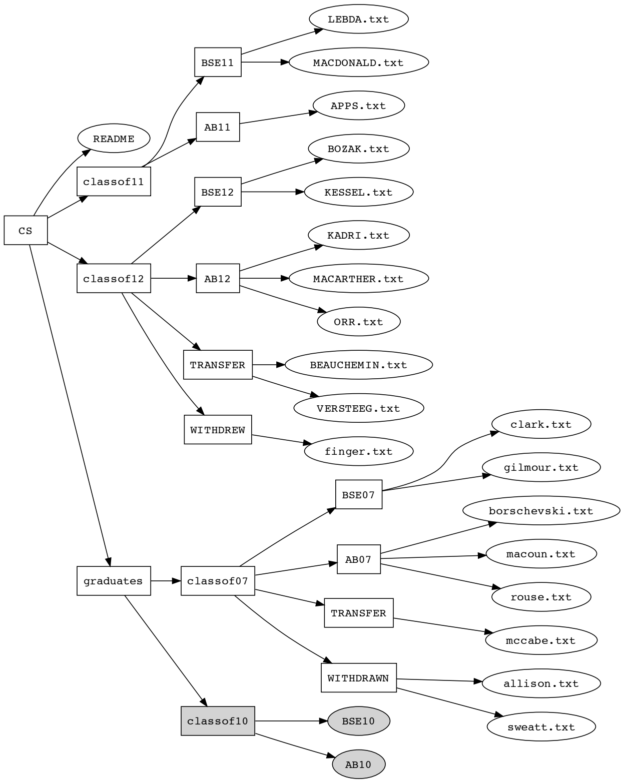

Figure 1. Anonymized snippet of Princeton computer science undergraduate data. A shaded node denotes denotes an error; in this case, missing files.

three directories: classof11 (seniors), classof12 (juniors) and graduates (students who have graduated). There is also a README file containing a collection of notes. Inside graduates, there is set of directories named classofYY where YY dates back to 92. Inside each classofYY directory, there are at least the two degree subdirectories ABYY and BSEYY as the computer science department gives out both Arts and Science (AB) and Engineering (BSE) degrees. Optionally, there are also subdirectories for students who withdrew from Princeton or transferred to another program. Within any degree subdirectory, there is one text file per student that records the courses taken and the corresponding grades. Each degree directory may also contain a template file named sss.txt or sxx.txt for creating new students.

The second filestore contains log files for CoralCDN [11, 12]. To monitor the performance and security of the system, the hosts participating in CoralCDN periodically send usage statistics back to a central server. These statistics are collected in a filestore similar to the one depicted in Figure 2. The filestore has a top-level directory named dat, which contains a set of subdirectories, one for each host. Each of those directories contain another set of directories, labeled by date and time. Finally, each of the date/time directories contain one or more compressed log files. For the purposes of this example, we will focus on the coralwebsrv.log.gz log file, which contains detailed information about the web requests made on the host during the preceding time period. In addition to exploring this primary filestore, we also explore a secondary, derived filestore. This secondary store, named stats, contains files that store statistics generated by Forest/Haskell scripts that analyze and summarize the raw CoralCDN server data. These system-wide summaries are representative of the statistics reported by Freedman in his CoralCDN report [11].

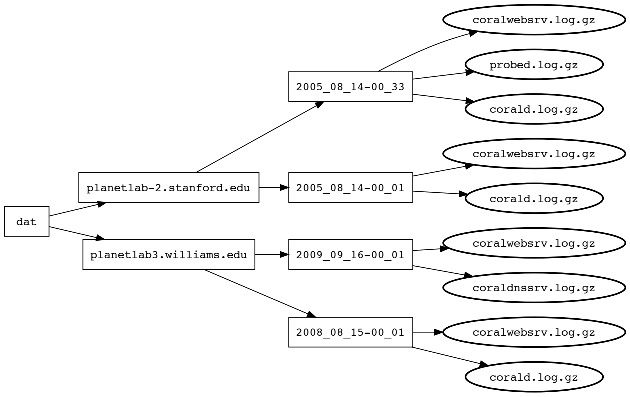

Figure 2. Coral system log data.

3. Forest Design

Data stored in filestores shares many characteristics of data stored in ordinary, in-memory data structures. Consequently, Forest uses the same sort of language to describe filestores as one uses to describe ordinary data structures — the language of types. Simple base types describe individual file system objects[1] and more complex types describe organized collections of file system objects. This idea forms the basis for our design.

Embedding Forest in Haskell. In order to write lightweight scripts, programmers must be able to manipulate and transform file system objects side-by-side with ordinary data structures. Consequently, a language like Forest must be embedded within a more general host programming language. We chose Haskell as the host language primarily because of its rich support for type-directed programming, which facilitates the construction of generic tools that can operate over any Forest description. As a bonus, Haskell's quasiquoting mechanism [21] proved a useful way to implement Forest. It enabled tight integration of the two languages while admitting fine-grained control over Forest syntax.

To introduce new Forest declarations within a Haskell program, the programmer simply opens the Forest sublanguage using quasiquoting notation:

```
[forest| ... forest declarations ... |]
```

When processing such a quasiquote, the Haskell compiler invokes the Forest compiler, which converts the given Forest declarations into a sequence of plain Haskell declarations that collectively implement the Forest declarations.

Forest Structure and Interpretations. Once within the Forest sublanguage, the programmer writes declarations that resemble extended Haskell type declarations. Each such type declaration has three primary semantic interpretations:

1. An interpretation as an *expected on-disk shape* of a file system fragment.
2. An interpretation as an ordinary Haskell type for the in-memory *representation* that will be constructed when the file system fragment is loaded into a Haskell program.
3. An interpretation as an ordinary Haskell type for the in-memory *metadata* that will be generated when the file system fragment is loaded.

All three interpretations are used by the tool that loads data from the filestore into memory as specified by a Forest description. When

[1] We use the term *file system object* or more simply *object* to denote either a file, a directory, or a symbolic link.

supplied with a *current path*, the loader uses the first interpretation to validate that the filestore rooted at that path has the correct shape. If the expected shape is complicated, possibly involving several nested subshapes (and hence traversal through several subdirectories), the semantics of Forest dictates how the loader should adjust the current path as it goes. When validation (also called *matching*) succeeds, we say the filestore fragment *matches* the description. The second interpretation is used when the loader lazily pulls the on-disk data into memory. The in-memory data structure is guaranteed to have the Haskell type given by the second interpretation. The third interpretation provides a type for the metadata structure generated by the loader. Such metadata includes error information (missing file, insufficient permissions, *etc.*) as well as file system attributes (owner, size, *etc.*).

The effectiveness of the Forest language comes in part from the fact that these three interpretations all arise from a single compact description. Moreover, to aid the programmer in navigating between interpretations, we align the syntax of Forest with the syntax of Haskell where possible. For example, if the Haskell types for the in-memory representation and metadata are record types, then the Forest syntax is designed to look similar to a Haskell record type. Likewise, if the Haskell types for the in-memory representation and metadata are Maybe types then the Forest syntax is designed to look similar to a Haskell Maybe type. Many of these high-level design considerations were adopted from earlier work on Pads [5, 7, 22], although the semantics of Forest (which operates over graph-based filestores) is substantially different from the semantics of Pads (which operates over sequence-based strings).

Errors. As with Pads [5, 7, 22], we do not assume that a given filestore conforms perfectly to its associated Forest description. Instead, when loading data, we check that a filestore conforms and mark discrepencies in the metadata. This design allows users to respond in application-specific ways to errors. It also allows Forest to check the arbitrarily complex conditions that may be specified by Forest's dependent types.[2] Because Forest loads data lazily, this choice means errors will not be detected unless the user program needs to touch the portion of the filestore with the error. The user can force a complete conformance check by accessing the top-level error count. It is possible for the filestore to change during or after this check. For the filestores we have seen in practice, there are extra-linguistic procedures in place to prevent such concurrent modifications; we leave to future work the possibility of using operating system support to monitor and/or prevent such changes automatically.

Onward. In the remainder of this section, we discuss the specific type constructors that constitute the Forest language and illustrate their use in our running examples.

3.1 Base Types: Files

Forest provides a small collection of base types for describing individual files: `TextFile` for ASCII files, `BinaryFile` for binary files, and `AnyFile` for arbitrary files. As with all Forest types, each of these types specifies a representation type, a metadata type, and loading and storing functions. For all three file types, the representation type is a `ByteString`. Similarly all three share a metadata type, which pairs file-system metadata with metadata describing properties of the file contents. The file-system metadata has type `Forest_md`, shown in Figure 3. This structure stores two kinds of information:

1. the number and kind of any errors that occurred during loading
2. the attributes associated with the file (`fileInfo`)

```
data Forest_md = Forest_md
   { numErrors :: Int
   , errorMsg  :: Maybe ErrMsg
   , fileInfo  :: FileInfo }

data FileInfo = FileInfo
   { fullpath    :: FilePath
   , owner       :: String
   , group       :: String
   , size        :: COff
   , access_time :: EpochTime
   , mod_time    :: EpochTime
   , read_time   :: EpochTime
   , mode        :: FileMode
   , isSymLink   :: Bool
   , kind        :: FileType }
```

Figure 3. Forest metadata types.

File-content metadata describes errors within the file. For these three file types, there is no meaningful content metadata and so this type is the unit type. Leveraging Haskell's laziness, the loading functions create the in-memory representations and set the metadata on demand. The storing functions, which are described in more detail in Section 4, do the inverse.

Although useful, these three base types are not sufficient for describing the wide range of files used in practice, including XML documents, Makefiles, source files in various languages, shell scripts, *etc.* The appropriate representation and content metadata types for each such file varies. To support such files, Forest provides a plug-in architecture, allowing third-party users to define new file types by specifying a representation type, a metadata type, and corresponding loading and storing functions.

A common class of files are *ad hoc data files* containing semi-structured information, an example of which is the Princeton student record file format. In such cases, Forest can leverage the Pads/Haskell [8] data description language to define format-specific in-memory representations, content metadata, and loading and storing functions. Pads/Haskell is a recently developed version of Pads [5, 7, 22]. Like Forest, Pads/Haskell is embedded in Haskell using quasiquotation. For example, the following code snippet begins the Pads specification of the Princeton student record format:

```
[pads| data Student(name::String) = Student
            { person  :: Line (Person name)
            , Header
            , courses :: [Line Course]
            , Trailer
            }
        ...  |]
```

This description is parameterized by the name of the student whose data is in the file; the complete description appears in the companion technical report [4]. From this specification, the Pads compiler generates an in-memory representation type `Student`, a content metadata type `Student_md`, and parsing and printing functions.

Forest provides the `File` type constructor to lift Pads types to Forest file types. For example, the declaration

```
[forest| type SFile(n::String) = File(Student n) |]
```

introduces a new file type named `SFile` whose format is given by the Pads type `Student`. As with the Pads type, `SFile` is parameterized by the name of the student.

[2] Validation that a filestore obeys a Forest specification is akin to type checking. However, it is akin to type checking closed, zero-order values (trees and graphs) as opposed to type checking parameterized, higher-order values (functions). Consequently, even though Forest has dependent types, type checking is not algorithmically challenging. For example, Forest does not have to decide equivalence of expressions with free variables as one must do when type checking a dependent lambda calculus.

Using Pads/Haskell descriptions in Forest not only helps specify the structure of ad hoc data files, but it also generates a structured in-memory representation of the data, allowing Haskell programmers to traverse, query and otherwise manipulate such data. We designed Pads/Haskell and Forest to work seamlessly together. From the perspective of the Haskell programmer traversing a resulting in-memory data structure, there is effectively no difference between iterating over files in a directory or structured sequences of lines or tokens within a file.

While Pads/Haskell is independently interesting, this paper focuses on Forest. Henceforth, any unadorned declarations occur within the Forest scope `[forest|...|]` unless otherwise noted. Any declarations prefixed by > are ordinary Haskell declarations.

3.2 Base Type: Symbolic Links

When symbolic links occur in a described filestore, Forest follows the symbolic link to its target, mimicking standard shell behavior. However, Forest allows programmers to specify explicitly that a particular file is a symbolic link using the base type `SymLink`. The in-memory representation for an explicit symbolic link is the path that is the target of the link. It is possible to use constraints (Section 3.6) to specify desired properties of the link target, such as requiring it to be to a specific file.

In Forest, any file system object may be described in multiple ways. Hence, in the case of a symbolic link, it is possible to use one declaration to specify that the object is a symbolic link and a second to specify the type of the link target. We will see an example of such a specification in Section 3.4

3.3 Maybe: Optional File System Objects

Sometimes, a given file (or directory or symbolic link) may or may not be present in the file system, and either case is valid. To handle this situation, we leverage the idea of an option type by providing a Forest-level `Maybe` type constructor that maps the optional file system object to a `Maybe` type in Haskell. In particular, if `T` is a Forest type, then `Maybe T` is the Forest type denoting an optional `T`. The type `Maybe T` succeeds and returns representation `None` when the current path does not exist in the file system. `Maybe T` also succeeds and returns `Just v` for some `v` of type `T` when the current path exists and matches `T`. `Maybe T` registers an error in the metadata when the current path exists but the corresponding object does not match `T`.

3.4 Records: Directories

Forest directories are record-like datatype constructors that allow users to specify directory structures. For example, to specify the root directory of the student repository in Figure 1, we might use the following declaration. This declaration assumes that we have already defined `Class y`, a parameterized description that specifies the structure of a directory holding data for the class of year `y`, and `Grads`, a description that specifies the structure of the directory holding all graduated classes.

```
type PrincetonCS_1 = Directory
  { notes   is "README"    :: TextFile
  , seniors is "classof11" :: Class 11
  , juniors is "classof12" :: Class 12
  , grads   is "graduates" :: Grads }
```

Each field of the record describes a single file system object. It has three components: (1) an internal name (*e.g.*, `notes` or `seniors`) that must be a valid Haskell record label, (2) an external name specified as a value of type `String` (*e.g.*, `"README"` or `"classof11"`) that gives the name of the object on disk, and (3) a Forest description of the object (*e.g.*, `TextFile` or `Class 11`).

When the external name is itself a valid Haskell label, users may omit it, in which case Forest uses the label as the on-disk name:

```
type PrincetonCS_2 = Directory
  { notes is "README" :: TextFile
  , classof11 :: Class 11
  , classof12 :: Class 12
  , graduates :: Grads }
```

We could not abbreviate the `notes` field because labels must start with a lowercase letter in Haskell.

Matching. For a file system object to match a directory description, the object must be a directory and each field of the record must match. A field `f` matches when the object whose path is the concatenation of the current path and the external name of `f` matches the type of `f`.

It is possible for the same file system object to match multiple fields in a directory description at the same time. For example, if `"README"` were actually a symbolic link, it is possible to document that fact by mentioning it twice in the directory description, once as a text file and once as a symbolic link:

```
type PrincetonCS_3 = Directory
  { link  is "README" :: SymLink
  , notes is "README" :: TextFile
  , ... }
```

It is also possible for a directory to contain objects that are unmatched by a description. We allow extra items because it is common for directories to contain objects that users do not care about. For example, a directory structure may contain extra files or directories related to a version control system, and a description writer may not want to clutter the Forest specification with that information. We will see shortly that it is possible to specify the absence of file system objects using constraints.

As suggested by the syntax, the in-memory representation of a directory is a Haskell record with the corresponding labels. The type of each field is the representation type of the Forest type for the field. The metadata has a similar structure. The metadata for each field has two components: file-system attribute information of type `Forest_md` and field-specific metadata whose type is derived from the Forest type for the field. In addition, the directory metadata contains an additional value of type `Forest_md` that summarizes the errors occurring in directory components and stores the `FileInfo` structure for the directory itself. When loading a directory, Forest constructs the appropriate in-memory representation for each field that matches and puts the corresponding metadata in the metadata structure. For fields that do not match, Forest constructs default values and marks the metadata with suitable error information.

Computed Paths The above descriptions are a good start for our application, but they are not ideal. Every year, the directory for graduating seniors (*i.e.*, `classof11`) is moved into the graduates directory, the juniors are promoted to seniors and a new junior class is created. As it stands, we would have to edit the description every year. An alternative is to parameterize the description with the current year and to *construct* the appropriate file names using Haskell functions:

```
> toStrN i n = (replicate(n - length(show i)) '0')
>              ++ (show i)
> mkClass y = "classof" ++ (toStrN y 2)

type PrincetonCS (y::Integer) = Directory
  { notes   is "README" :: TextFile
  , seniors is <|mkClass y    |> :: Class y
  , juniors is <|mkclass (y+1)|> :: Class <|y+1|>
  , graduates :: Grads }
```

The bracket syntax `<|...|>` provides an escape so that we may use Haskell within Forest code to specify arbitrary computations. When an expression is a constant or variable, it may be supplied directly. When an argument is more complex, however, it must be written in brackets to escape to Haskell. This example also illustrates abstraction: any Forest declaration may be parameterized by specifying a legal Haskell pattern and its type. The types of the fields for `seniors` and `juniors` illustrate the use of parameterized descriptions.

Approximate Paths As filestores evolve, naming conventions may change. Additionally, directory structures with multiple instances may have minor variations in the names of individual files across instances. For example, in each Princeton class directory, there may (or may not) be some number of students that have withdrawn from the program, transferred to a different program, or gone on leave. Over the years, slightly different directory names have been used to represent these situations.

To accommodate this variation, Forest includes the matching construct to approximate file names. We can use this mechanism to describe the class directory:

```
> transRE = RE "TRANSFER|Transfer"
> leaveRE = RE "LEAVE|Leave"
> wdRE    = RE "WITHDRAWN|WITHDRAWAL|Withdrawn"

type Class (y::Integer) = Directory
  { bse is <|"BSE" ++ (toString y)|> :: Major
  , ab  is <|"AB"  ++ (toString y)|> :: Major
  , trans matches transRE :: Maybe Major
  , withd matches wdRE    :: Maybe Major
  , leave matches leaveRE :: Maybe Major }
```

A field with the form `<label>` **`matches`** `<regexp> :: T` finds the set of paths in the files system that match `currentPath/<regexp>`. If there are zero or one such files, the `matches` form acts just as the `is` form. If more than one file matches, one of the matches is selected non-deterministically, a multiple match error is registered in the metadata, and matching continues as it would with the `is` form. In addition to regular expressions, the matching construct also allows *glob patterns*, (*i.e.*, patterns such as `*.txt`), to specify the names of files on disk. An example appears in the next subsection.

3.5 Lists

Just as Haskell has both records and lists, so too does Forest. Records allow programmers to specify a fixed number of file system objects, each with its own type. Lists, on the other hand, allow programmers to specify an arbitrary number of file system objects, each with the same type. As an example, we can use a list to specify the `Grads` directory from Figure 1. We borrow Haskell's notation for list comprehensions to specify the names of the file system objects:

```
> getYear s =
>   toInteger $ reverse $ take 2 $ reverse s
> cRE = RE "classof[0-9][0-9]"

type Grads =
  [c :: Class <|getYear c|> | c <- matches cRE]
```

In this specification, `Grads` is a directory fragment containing a number of `Class` subdirectories with names `c` that match the regular expression `cRE`. The Haskell function `getYear` extracts the last two digits from the name of the directory, converts the string digits to an integer year, and passes the year to the underlying `Class` specification. More generally, Forest lists have the form `[path :: T | id <- gen, pred]` where `id` is bound in turn to each of the file names generated by `gen`, which may be a `matches` clause (used to match against the files at the current path as in the previous section) or a list computed in Haskell. These generated `id`s are filtered by the optional predicate `pred`. For each such legal `id`, there is a corresponding expression `path`, which Forest interprets as extending the current path. The object at each such path should have the Forest type `T`. The identifier `id` is in scope in `pred`, `path`, and `T`.

The in-memory representation of a Forest list is a Haskell list containing pairs of the name of a matching object and its representation. The metadata is a list of the metadata of the matching objects paired with a summary metadata structure of type `Forest_md`.

Representation Transformations. Although the list representation for comprehensions is useful, it can be desirable to use a more sophisticated data structure to represent such collections. To support this usage, Forest allows programmers to prefix a list comprehension with any type constructor that belongs to a Forest-defined container type class. This type class contains functions that specify how to convert between the list representation and the desired container representation. We have provided such instance declarations for Haskell's `Map` and `Set` type constructors.

As an example, consider the specification of the `Major` directory. Each such directory contains a list of student files and an additional template file named either `sss.txt` or `sxx.txt`. The declaration below specifies the collection of student files by matching with a glob pattern and filtering to exclude template files. It uses the `Map` type constructor to specify that the data and metadata should be collected in a `Map` rather than a list.

```
> template s = s `elem` ["sss.txt", "sxx.txt"]
> txt = GL "*.txt"

type Major = Map
   [ s :: File (Student <|dropExtension s|>)
   | s <- matches txt, <|not (template s)|>]
```

3.6 Dependent Types: Attributes and Constraints

Every file system object has a number of *attributes* associated with it, such as its owner, group, permissions, and size. In general, if a Forest identifier `id` refers to a path, then the identifier `id_att` refers to the corresponding attributes. This attribute identifier has type `Forest_md`, shown in Figure 3. Forest defines helper functions to access these attributes, some of which are listed in Figure 4.

Constrained types are a simple form of dependent types that allow users to specify required attributes. For example, the type `PrivateFile` specifies a text file accessible only by its owner.

```
type PrivateFile = TextFile
      where <|get_modes this_att == "-rw-------"|>
```

The keyword **`where`** introduces a constraint on the underlying type. The load function for the type `PrivateFile` checks this constraint during loading. If the constraint is false, it records that fact in the metadata. Within constraints, the special identifier `this` refers to the representation of the underlying object, `this_att` refers to its attributes and `this_md` to its complete metadata.

Using attributes, we can write a *universal directory description*, which is sufficiently general to describe any directory:

```
type Universal = Directory
  { asc is [ f :: TextFile
           | f <- matches <| GL "*" |>,
          <| get_kind f_att == AsciiK |> ]
  , bin is [ b :: BinaryFile
           | b <- matches <| GL "*" |>,
          <| get_kind b_att == BinaryK |> ]
  , dir is [ d :: Universal
           | d <- matches <| GL "*" |>,
          <| get_kind d_att == DirectoryK |> ]
  , sym is [ s :: SymLink
           | s <- matches <| GL "*" |>,
          <| isJust (get_symLink s_att) |> ] }
```

function name	information
`get_group`	object group
`get_kind`	the sort of file or directory
`get_modes`	permission string
`get_owner`	object owner
`get_size`	object size

Figure 4. Selected file attribute functions.

When a directory is loaded using the `Universal` description, all the ASCII files will end up the in `asc` field, all the binary files in `bin`, all the directories in `dir`, and all the symbolic links in `sym`. Note that the description uses recursion to describe nested directories. In the case that a symbolic link creates a cycle in the file system by pointing to a parent directory, the Haskell in-memory representation is a (lazy) infinite data structure. We view the fact that it is possible to write such a universal description in Forest as evidence that the language is appropriately expressive. This description also serves as an example of how to describe a filestore by its *structure* rather than its *names*.

We can also use constraints to specify that certain files *do not* appear in certain places. As an example, we might want to require that no binaries appear in a directory given to an untrusted user as scratch space. The description below flags an error during loading if a binary file exists in the directory.

```
type NoBin =
 [ b :: BinaryFile | b <- matches <| GL "*" |>,
                <| get_kind b_att == BinaryK |> ]
  where <|length this == 0|>
```

3.7 Specialized Constructors: Gzip and Tar

Some files need to be processed before they can be used. A typical example is a compressed file such as the gzipped log files in CoralCDN. Forest provides processing-specific type constructors to describe such files. For example, if `CoralLog` is a Pads/Haskell description of a CoralCDN log file then

```
type Info = Gzip (File CoralLog)
```

describes a gzipped log file. Likewise, suppose `logs.tar.gz` is a gzipped tar file and that the type `ManyLogs` describes the directory of log files that `logs.tar` expands to when untarred. Such a situation can be described using a combination of the `Tar` and `Gzip` type constructors:

```
type MoreInfo = Gzip (Tar ManyLogs)
```

3.8 Putting it all together

The preceding subsections give an overview of Forest's design. Figures 5 and 6 give the specifications for the two running examples, minus the associated Pads/Haskell and Haskell declarations. The complete descriptions of these filestores and additional descriptions are available in a technical report [4], including descriptions of the Pads website, a Gene Ontology filestore, and CVS repositories.

4. Programming with Forest

Many Forest programs work in two phases. In the first phase they use Forest to load relevant portions of the file system into memory, and in the second phase they use an ordinary Haskell function to traverse the in-memory representation of the data (or its associated metadata) and compute the desired result. Some Forest programs add a third phase in which they store updated structures back to the filestore.

```
[forest|
 data PrincetonCS (y::Integer) = Directory
   { notes   is "README" :: TextFile
   , seniors is <|mkClass y    |> :: Class y
   , juniors is <|mkclass (y+1)|> :: Class <|y+1|>
   , graduates :: Grads }

 data Class (y::Integer) = Directory
   { bse is <|"BSE" ++ (toString y)|> :: Major
   , ab  is <|"AB"  ++ (toString y)|> :: Major
   , trans matches transRE :: Maybe Major
   , withd matches wdRE    :: Maybe Major
   , leave matches leaveRE :: Maybe Major }

 type Grads =
   [c :: Class <|getYear c|> | c <- matches cRE]

 type Major = Map
    [ s :: File (Student <|dropExtension s|>)
    | s <- matches txt, <|not (template s)|>] |]
```

Figure 5. Forest description of Princeton filestore.

```
[forest|
 type Stats = Directory
   { last :: File Last, topk :: File Topk }
 type Dat   = [ s :: Site | s <- matches site ]
 type Site  = [ d :: Log  | d <- matches time ]
 data Log = Directory
   { log is coralwebsrv :: Gzip (File CoralLog) } |]
```

Figure 6. Forest description of CoralCDN filestore.

Representation Types:

```
newtype Stats = Stats {last :: Last, topk :: Topk}
newtype Dat   = Dat   [(String, Site)]
newtype Site  = Site  [(String, Log)]
data    Log   = Log   {log :: CoralLog}
```

Metadata Types:

```
type Stats_md = (Forest_md, Stats_inner_md)
data Stats_inner_md = Stats_inner_md
    {last_md :: (Forest_md, Last_md),
     topk_md :: (Forest_md, Topk_md)}
type Dat_md   = (Forest_md, [(String, Site_md)])
type Site_md  = (Forest_md, [(String, Log_md)])
type Log_md   = (Forest_md, Log_inner_md)
data Log_inner_md = Log_inner_md
    {log_md :: (Forest_md, CoralLog_md)}
```

Load Functions:

```
stats_load :: FilePath -> IO (Stats, Stats_md)
dat_load   :: FilePath -> IO (Dat, Dat_md)
site_load  :: FilePath -> IO (Site, Site_md)
log_load   :: FilePath -> IO (Log, Log_md)
```

Store Functions:

```
stats_manifest :: (Stats, Stats_md) -> IO Manifest
dat_manifest   :: (Dat, Dat_md)     -> IO Manifest
site_manifest  :: (Site, Site_md)   -> IO Manifest
log_manifest   :: (Log, Log_md)     -> IO Manifest

storeAt        :: FilePath -> Manifest -> IO ()
store          ::             Manifest -> IO ()
```

Figure 7. Artifacts generated from the CoralCND description.

To facilitate this style of programming, the Forest compiler generates several Haskell types and functions from every Forest declaration. Collectively, these types and functions define an instance of the `Forest` type class:

```
class (Data rep, ForestMD md)
      => Forest rep md | rep -> md  where
   load     :: FilePath -> IO(rep, md)
   manifest :: (rep,md) -> IO Manifest
   ...
```

In this type class, the type `rep` is the generated in-memory representation type of the corresponding on-disk data. The type `md` is the generated type for the associated metadata. The `ForestMD` type class provides operations for manipulating Forest metadata; all generated metadata types belong to this type class.

The generated load function lazily traverses the file system and reads the files, directories, and symbolic links mentioned in the description into a pair of the in-memory representation and its metadata. To reverse the process of reading data in to memory, Forest also generates a *manifest function*, which reads an in-memory data structure, writes its contents out to disk in a temporary space, and prepares a *manifest log*. The manifest log records inconsistencies detected during this process as well as the sequence of operations necessary to move data from the temporary space to its final resting point. Inconsistences can arise when a programmer creates an erroneous in-memory representation of a filestore. The dependencies that may be present in Forest descriptions mean that not all such inconsistencies can be detected statically by the Haskell type system. After creating a manifest, a programmer may analyze it and decide whether to execute the generic `store` or `storeAt` functions, which move a manifest (inconsistencies and all) to its rightful position on disk. Details concerning the semantics of storing, especially where it concerns inconsistencies, are explained in further depth in Section 7.

As an example, consider the CoralCDN logs described in Figure 6. The corresponding load and store functions, the representation types, and the metadata types appear in Figure 7.[3] Note that the structure of each of these artifacts mirrors the structure of the Forest description that generated them. This close correspondence makes it easy for programmers to write programs using these Forest-generated artifacts.

For instance, consider the `Dat` description in Figure 6. The `dat_load` function takes a path as an argument and produces the representation and metadata obtained by loading each of the site directories contained in the directory at that path:

```
(rep,md) <- dat_load "/var/log/coral/dat"
```

Because `Dat` is a Forest list, the `rep` is a Haskell list. More specifically, `rep` has the form

```
Coral [("planetab2.eecs.wsu.edu", Site [...]),
       ("planetlab3.williams.edu",Site [...]),...]
```

where the list contains pairs of names of subdirectories and representations for the data loaded from those directories. The metadata is a pair consisting of a generic header of type `Forest_md` and a list of pairs of names of subdirectories and their associated metadata. The header collects information about errors encountered during loading and it stores the file system attributes of each file, directory, or symbolic link loaded from the file system. The following is the pretty-printed version of such a structure:

```
Forest_md
  { numErrors = 0,
    errorMsg = Nothing,
    fileInfo = FileInfo
      { fullpath = /var/log/coral/dat,
        owner = alice, group = staff, size = 102,
        access_time = Fri Nov 19 01:47:09 2010,
        mod_time = Thu Nov 18 20:42:37 2010,
        read_time = Fri Nov 19 01:47:28 2010,
        mode = drwxr-xr-x, isSymLink = False,
        kind = Directory } },
[("planetlab2.eecs.wsu.edu", Forest_md {...}),
 ("planetlab3.williams.edu", Forest_md {...}), ...]
```

Using these functions and types, it is easy to formulate many useful queries as simple Haskell programs. For instance, to count the number of sites we can simply compute the length of the nested list in `rep`:

```
num_sites = case rep of Dat l -> List.length l
```

More interestingly, since the internals of the web log are specified using Pads/Haskell (see the technical report [4] for details), it is straightforward to dig in to the file data and combine it with file metadata or attributes in queries. For example, to calculate the time when statistics were last reported for each site, we can zip the lists in `rep` and `md` together and project out the site name and the `mod_time` field from each element in the resulting list of pairs:

```
get_site = fst
get_mod (_,(f,_)) = mod_time . fileInfo $ f
sites_mod () =
  case (rep,md) of (Dat rs, (_,ms)) ->
    map (get_site *** get_mod) (zip rs ms)
```

As this example shows, Forest blurs the distinction between data represented on disk and in memory. After writing a suitable Forest description, programmers can write programs that work on file system data as if it were in memory. Moreover, because Forest uses Haskell's lazy I/O operations, many simple programs do not require constructing an explicit representation of the entire directory being loaded in memory—a good thing as the directory of CoralCDN logs contains approximately 1GB of data! Instead, the load functions only read the portions of the file system that are needed to compute the result—in this case, only the site directories and not the gzipped log files contained within them.

As a final analysis example, consider a program that computes the top-k requested URLs from all CoralCDN nodes by size. The CoralCDN administrators compute this statistic periodically to help monitor and tune the performance of the system [11]. We define the analogous function in Haskell using helper functions such as `get_sites` to project out components of `rep`:

```
topk k =
  take k $ sortBy descBytes $ toList $
  fromListWith (+)
    [ (get_url e, get_total e)
    | (site,sdir) <- get_sites rep,
      (datetime,ldir) <- get_dates sdir,
     e <- get_entries ldir,
     is_incoming e ]
```

Reading this program inside-out, we see that it first uses a list comprehension to iterate through `rep`, collecting the individual log entries corresponding to incoming requests using the `is_incoming` predicate. It then projects out the URL requested and the total size of the request. It then sums the sizes of all requests for the same URL using the `fromListWith` function from the `Data.Map` module. Next, it sorts the entries in descending order. Finally, it returns the first k entries of the list as the final result.

Having implemented these analyses, a programmer may wish to store their results. She may do so via the following code, which

[3] In the following examples, for the sake of clarity, we use type-specific names such as `dat_load` and `dat_manifest`, rather than the overloaded names `load` and `manifest`.

uses stats_manifest to generate a manifest and store to copy it over to the stats directory. In addition, the code uses stats_defaultMd, a function that constructs default metadata for stats structures (a useful function in situations that require storing newly constructed data).

```
let result = Stats { last = sites_mod ()
                   , topk = topk 10 }
manifest <- stats_manifest
  ( result
  , stats_defaultMd result "/var/log/coral/stats" )
store manifest
```

Overall, the main thing to take away from this section is how Forest and its tight integration with Haskell facilitates exploratory data analysis, enabling remarkably terse queries over the combination of file contents, file attributes and directory structures.

5. Generic Tools

Third-party developers can use generic programming [18] to generate tools that will work for any filestore that has a Forest description. An advantage of these tools compared to tools that work directly on the untyped file system is that they are specific to the fragment of the file system relevant to the filestore. This fragment can be difficult to specify when using conventional tools since it can rely on the contents of configuration files, file naming conventions, file system attributes, *etc.* It is precisely these relationships that Forest descriptions capture concisely; tools written to use Forest specifications can leverage that information.

As a proof of concept, we have written a number of such tools, which we describe in this section.

5.1 Generic Querying

One simple application of generic programming is querying metadata to find files with a particular collection of attributes. The findFiles function

```
findFiles :: (ForestMD md) =>
     md -> (FileInfo -> Bool) -> [FilePath]
```

takes as input any Forest metadata value (*i.e.*, any value of type md where md belongs to the Forest metadata class ForestMD) and a predicate on FileInfo structures, and returns the list of all FilePaths anywhere in the input metadata whose associated FileInfo satisfies the predicate. For example, if cs_md is the metadata associated with the Princeton computer science department filestore, then the code

```
dirs  = findFiles cs_md (\(r::FileInfo) ->
                           (kind r) == DirectoryK)
other = findFiles cs_md (\(r::FileInfo) ->
                           (owner r) /= "bwk")
```

binds dirs to the list of all directories in the data set and other to all the directories and files not owned by user "bwk".

To implement the findFiles function, we use the generic Haskell function listify:

```
findFiles md pred = map fullpath (listify pred md)
```

The return type of the polymorphic listify function is instantiated to match the argument type of its predicate argument. We map the fullpath function over the resulting list of FileInfo structures to return only the FilePaths.

5.2 File System Visualization

ForestGraph generates a graphical representation of any directory structure that matches a Forest specification. We generated the graphs in Figures 1 and 2 using this tool. In the default configuration, ForestGraph uses boxes to denote directories and ovals to denote files. Borders of varying thickness distinguish between ASCII and binary files. Dashed node boundaries indicate symbolic links and shaded nodes flag errors.

The core functionality of ForestGraph lies in the Haskell function mdToPDF:

```
mdToPDF :: ForestMD md =>
     md -> FilePath -> IO (Maybe String)
```

The function takes as input any metadata value and a filepath that specifies where to put the generated PDF file. It optionally returns a string (Maybe String); if the option is present, the string contains an error message. The IO type constructor indicates that there can be side effects during the execution of the function. A use of this function to generate the graph for the Princeton computer science department filestore looks like:

```
 do { (cs_rep,cs_md) <- CS_load  "facadm"
    ; mdToPDF cs_md "Output/CS.pdf"       }
```

Note that this code needs only the metadata to generate the graph; laziness means Forest will not load the representation in this case.

The related function mdToPDFWithParams takes an additional argument that allows the user to specify how to draw the nodes and edges in the output graph. Among other things, this parameter specifies how to map a value of type Forest_md into GraphViz [13, 14] attributes. By appropriately setting the parameter, a user can customize the formatting of each node according to its owner, group, or permissions, *etc.*, as well as specify global properties of the graph such as its orientation and size. ForestGraph uses the Haskell binding of the GraphViz library to lay out and render the graphs, so all customizations provided by GraphViz are available.

The listify function is at the heart of the implementation of this tool; we use it to convert the input metadata to the list of FileInfos in the metadata. We then convert this list into a graph data structure suitable for use with the GraphViz library.

5.3 Permission Checker

The permission tool is designed to check the permissions on the files and directories in a Forest description on a multi-user machine. In particular, it enables one user to determine which files a second user can read, write, or execute. If the second user cannot access a file in a particular way, the tool also reports the names of the files and directories whose permissions have to change to allow the access. The tool is useful when trying to share files with a colleague. It helps the first user ensure that all the necessary permissions have been set properly to allow the second user access. The key to the implementation of this tool is again applying the listify function to the metadata for the Forest description.

5.4 Shell Tools

We have implemented analogs of many shell tools that work over a file system fragment defined by a Forest description:

```
ls   :: (ForestMD md) => md -> String -> IO String
grep :: (ForestMD md) => md -> String -> IO String
tar  :: (ForestMD md) => md -> FilePath -> IO ()
cp   :: (ForestMD md) => md -> FilePath -> IO ()
```

All of these functions work by extracting the relevant file names from the argument metadata structure using listify and then calling out to a shell tool to do the work. For ls, the second argument gives the command-line arguments to pass to the shell version of ls, and the result is the resulting output. The implementation uses xarg to lift the restriction on the number of files that can be passed to ls. For grep, the second argument is the search string and result is the output of the shell version of grep. For tar, the second argument specifies the location for the resulting tarball. The implementation uses a file manifest to allow `tar` to work regardless of the number of files involved. The `cp` tool uses the `tar` tool to move the files mentioned in the metadata to the location specified by the second argument *while retaining the same directory structure*. The module that implements these tools is approximately 80 lines of Haskell code.

5.5 Description Inference Tool

This tool allows the user to generate a Forest description from the contents of the file system. The function

```
getDesc :: FilePath -> IO String
```

takes as an argument the path to the root of the directory structure to infer. It returns a string containing the generated representation. For example, below we show a fragment of the results when `getDesc` is invoked on the `classof11` directory:

```
data classof11 = Directory {
    aB11 is "AB11" :: aB11,
    bSE11 is "BSE11" :: bSE11,
    tRANSFER is "TRANSFER" :: tRANSFER,
    wITHDREW is "WITHDREW" :: wITHDREW }
data tRANSFER = Directory {
    bEAUCHEMINtxt is "BEAUCHEMIN.txt" :: File Text,
    vERSTEEGtxt is "VERSTEEG.txt" :: File Text }
...
```

The description is not perfect: the label names are generated from the file name, for example. Nevertheless, the tool improves programmer productivity as it is easier for a programmer to edit a generated description than to start from scratch. Our first tool in this vein is simple; a more sophisticated variant would collapse records of files into lists when a width limit was exceeded or other criteria were met. Another variant might collapse deeply nested directories into a universal directory description when a depth limit was exceeded. The `getDesc` function works by using the universal description to load the contents of the file system starting from the supplied path. It then walks over the resulting metadata to generate a Forest parse tree, which it then pretty prints.

6. Implementation

The current implementation of Forest is available from the project web site: `forestproj.org`.

Haskell provides powerful language features and libraries that greatly facilitated implementation of Forest. The most obvious of these features is the quasiquotation mechanism [21] that we used to embed Forest into Haskell. This mechanism provided the benefits of being an embedded domain-specific language without having to sacrifice the flexibility of defining our own syntax. To use quasiquoting, we defined a Haskell value `forest` of type `QuasiQuoter` which specifies how to convert an input string representing a sequence of Forest declarations into the Template Haskell [27] data structures that represent the syntax of the corresponding collection of Haskell declarations. The Haskell compiler calls the `forest` "compilation" function during the compilation of any Haskell source file containing a Forest quasiquotation. The quasiquoted syntax `[forest| <input> |]` is legal anywhere the identifier `forest` is in scope. When the Haskell compiler processes this declaration, it first passes `<input>` as a string to the `forest` quasiquoter, and then it compiles the resulting Template Haskell data structures as if the corresponding Haskell code had appeared in the input at the location of the quasiquote. Early versions of quasiquoting supported quoting only expression and pattern forms. Simon Peyton Jones extended the mechanism to permit declaration and type quasiquoting partly to enable the Forest implementation. We used this same approach to implement Pads/Haskell, which we built concomitantly.

Note that in implementing Forest, we had to use Template Haskell rather than any of the other libraries that support generic programming, both because that is what the quasiquote library expects and because we need to generate type and datatype declarations (and to do so at compile time). Other available generic libraries do not support the latter functionality.

Parsing. We used the parsec 3.1.0 parser combinator library [19] to implement the Forest parser. One key element of the Forest design is to allow arbitrary Haskell expressions in various places inside Forest descriptions. We did not want to reimplement the grammar for Haskell expressions, which is quite complicated. Instead, we structured the Forest grammar so we could always determine the extent of any embedded Haskell code. We then used the Haskell Source Extension package [15] to parse these fragments. The data structure that this library returns is unfortunately not the data structure that Template Haskell requires, so we used yet another library, the Haskell Source Meta package [16], that does this translation.

Type checking. We would like to give users high-quality error messages if there are type errors in their Forest declarations. At the moment, typechecking occurs, but only after the Forest declarations have been expanded to the corresponding Haskell code. Although these error messages can be quite informative, it is sub-optimal to report errors in terms of generated code. Type checking the Forest source is complicated by the embedded fragments of Haskell. As with the syntax, we do not want to reimplement the Haskell typechecker! There is an active proposal [29] to extend the Template Haskell infrastructure with functions that would enable us to ask the native Haskell typechecker for the types of embedded expressions and to extend the current type environment with type bindings for new identifiers. With this combination of features, we would be able to type check Forest sources directly.

Runtime. Although Forest facilitates treating the file system as a persistent store, it does not provide the ACID guarantees familiar from databases. None of the filestores we have encountered in practice are implemented in a system that provides such support; users instead have extra-linguistic mechanisms to make sure they do not corrupt their data with ill-timed concurrent reads and writes. That said, the Forest language does not preclude an implementation from providing such guarantees. We consider this issue very interesting future work.

Forest uses Haskell's `unsafeInterleaveIO` to load each portion of a filestore only when needed by an application program. We have not systematically measured the performance overhead of using Forest. However, we have used our mostly-unoptimized implementation to manipulate filestores on the order of many gigabytes and found the performance acceptable for many applications.

The running time of storing operations is proportional to the "footprint" of the described filestore. However, the Forest compiler generates `load` and `manifest` functions for each named type in a description. Thus, updates may be made at any granularity for which there is a named type, which is typically at the level of individual files. We plan to investigate better support for incremental updates in future work.

7. Core Calculus

This section presents a core calculus for Forest, which formalizes the essential features of the language in a simple setting. The calculus is based on classical (*i.e.*, not separating, substructural, or ambient) unordered tree logics, but has a number of features tailored to file systems. We used it to study various features of Forest as we were developing it, and to prove key theorems like the round-tripping properties described at the end of the section.

Basic definitions

$$\begin{array}{rrl}
\textit{Integers} & n & \in \mathbb{Z} \\
\textit{Strings} & u & \in \Sigma^* \\
\textit{Values} & v & ::= n \mid u \mid r \mid \mathit{True} \mid \mathit{False} \mid () \mid (v_1, v_2) \\
 & & \quad \mid \mathit{Just}\ v \mid \mathit{Nothing} \mid [v_1, .., v_n] \\
\textit{Types} & \tau & ::= \mathit{String} \mid \mathit{Int} \mid \mathit{Path} \mid \mathit{Bool} \mid () \mid (\tau_1, \tau_2) \\
 & & \quad \mid \mathit{Maybe}\ \tau \mid [\tau] \\
\textit{Environments} & \mathcal{E} & ::= \emptyset \mid \mathcal{E}, x \mapsto v \\
\textit{Expressions} & e & ::= x \mid \lambda x.\ e \mid e_1\ e_2 \mid \ldots
\end{array}$$

Forest definitions

$$\begin{array}{rrl}
\textit{Paths} & r & ::= \cdot \mid r \,/\, u \\
\textit{Attributes} & a & ::= v \\
\textit{Contents} & T & ::= \mathsf{File}\ u \mid \mathsf{Link}\ r \mid \mathsf{Dir}\ \{u_1, .., u_n\} \\
\textit{File systems} & F & ::= \{\!|\, r_1 \mapsto (a_1, T_1), .., r_n \mapsto (a_k, T_n) \,|\!\} \\
\textit{Specifications} & s & ::= k_{\tau_1}^{\tau_2} \mid e{::}s \mid \langle x{:}s_1, s_2 \rangle \mid [s \mid x \in e] \mid \mathsf{P}(e) \mid s?
\end{array}$$

Figure 8. Forest calculus syntax.

7.1 Data Model

Forest calculus specifications manipulate instances of a simple file system model, which is given in Figure 8. A path r is a sequence of strings u,[4] and a file system F is a finite map from paths to pairs (a, T) of attributes a and file system contents T. We do not specify the syntax of attributes precisely, but expect that they will contain the usual fields including owner, group, permissions, modification date and time, *etc.* A special attribute a_{default} contains default values for all fields. The contents T of a node in the file system is either a file $\mathsf{File}\ u$, where u is the string contents of the file; a symbolic link $\mathsf{Link}\ r$, where r is the path pointed to by the link; or a directory $\mathsf{Dir}\ \{u_1, .., u_n\}$, where u_1 to u_n are the names of the elements of the file system located below the node. We write $\mathsf{dom}(F)$ for the set of paths F is defined on, $F(r)$ for the contents at r, $F(r) = \bot$ when r is not in $\mathsf{dom}(F)$, and $F[r := (a, T)]$ or $F[r := \bot]$ for the file systems obtained from F by overwriting the mapping for r to (a, T) or deleting the mapping for r respectively.

A file system F is *well-formed* if it encodes a tree with directories at the internal nodes and files and symbolic links at the leaves. More formally, F is well-formed if the following conditions hold:

- $\mathsf{dom}(F)$ is prefix-closed,
- $F(r) = (a, \mathsf{Dir}\ \{u_1, .., u_n\})$ implies $r \,/\, u_i \in \mathsf{dom}(F)$ for all i from 1 to n, and
- $F(r) = (a, \mathsf{File}\ u)$ or $F(r) = (a, \mathsf{Link}\ r')$ implies $r \,/\, u' \notin \mathsf{dom}(F)$ for all u'.

Note that although these conditions imply that the structure of a well-formed file system F must be tree-shaped, cycles can be expressed using symbolic links that point "up" in the file system.

7.2 Specifications

The syntax of specifications s is given in Figure 8. The calculus models the most important features of the full Forest language in the simplest possible way, using orthogonal, independent constructs. The set of specifications is parameterized over a collection of constants $k_{\tau_1}^{\tau_2}$, where τ_1 is the type of the generated representation and τ_2 is the type of the generated metadata. We omit the type annotations in examples. For the purpose of illustration, we will assume that constants for describing files File, directories Dir, and symbolic links Link all exist, as well as constants $\mathsf{Adhoc}(p)$ parameterized on Pads/Haskell descriptions p for describing the contents of files in more detail. Path specifications $e{::}s$ are parameterized on an expression e, which must evaluate to a string (to be appended to the current path), and a specification s that describes the fragment of the file system at the extended path. We leave the syntax of expressions e abstract but assume that it contains the features of a simple functional language (of course, in the full Forest language, expressions can be arbitrary fragments of Haskell code). We assume a semantic function $[\![e]\!]_\tau^{\mathcal{E}}$ that evaluates e in environment $\mathcal{E}$, yielding a value v of type τ. Dependent pairs are written $\langle x{:}s_1, s_2 \rangle$ where s_1 and s_2 are specifications that describe possibly overlapping fragments of the file system. When a dependent pair is evaluated, the variables x and x_{md} are bound to the value and metadata computed for s_1 and may be used in s_2. Comprehensions are written $[s \mid x \in e]$, where e is an expression that evaluates to a set of values, and s is a specification that describes a fragment of the file system for each value of x. Predicates $\mathsf{P}(e)$ succeed when the expression e evaluates to True and fail when it evaluates to False under the current environment. Finally, the specification $s?$ describes either a filesystem that is undefined at the current path, or a file system containing the current path and satisfying s.

To develop some intuitions about these constructs, let us encode a few of the more complicated features of the full Forest language in the calculus. In the full language, records and paths are specified using a single construct:

```
Directory
  { c is "c.txt" :: C,
    d is "d.txt" :: D c,
    e is "e.bin" :: E d }
```

The calculus, however, only has dependent pairs, not full-blown records, and has a separate construct for describing paths. Thus, to encode the specification above, we use the following specification:

$$\langle c : (\texttt{"c.txt"}{::}\mathrm{C})\ , \langle d : (\texttt{"d.txt"}{::}\mathrm{D}\ c)\ , \texttt{"e.bin"}{::}\mathrm{E}\ d \rangle\rangle$$

Similarly, a comprehension written as

```
[c :: C | c <- matches <| GL "*" |> ]
```

in the full language is encoded in the calculus as

$$\langle d{:}\mathsf{Dir}\ , [\ c{::}\mathrm{C}\ \mid c \in d\ ]\rangle$$

The value returned by the Dir constant is the set of names of the immediate children of the directory. Finally, in the full language, constraints are written as

```
s where e
```

but are encoded in the calculus as a dependent pair and predicate:

$$\langle \mathit{this}{:}s, \mathsf{P}(e)\rangle$$

When evaluated, the predicate encodes the success or failure of the constraint expressed by e in the metadata for the second component of the pair.

7.3 Semantics

The semantics of the Forest calculus is organized into four separate definitions, one for each of the major artifacts generated by the compiler. These artifacts include a type for the representations, a type for metadata, a function *load* for loading data from the file system, and a function *store* for storing it back. They are carefully designed to fit together in a particular way—*e.g.*, the *load* and *store* functions manipulate representations and metadata of

[4] For simplicity, we do not model special path elements such as ".." and ".". It would be easy to add these at the cost of complicating the semantics.

the appropriate type and are guaranteed (under certain precisely identified circumstances) to preserve data on round trips.

Types. The definitions of the types for the representations ($\mathcal{R}\llbracket s \rrbracket$) and metadata ($\mathcal{M}\llbracket s \rrbracket$) generated for each specification s appear in Figure 10. The representation type for constants $k^{\tau_2}_{\tau_1}$ is simply read off from τ_1. For other specifications, representation types are built out of the types for sub-specifications in the obvious way—*e.g.*, for pairs $\langle x{:}s_1, s_2\rangle$, the type is a product $(\mathcal{R}\llbracket s_1 \rrbracket, \mathcal{R}\llbracket s_2 \rrbracket)$. The types for metadata are more interesting. In the Forest calculus we represent metadata using structured datatypes containing a boolean value at each level of structure. Intuitively, this value indicates whether there were any errors during loading. The Md constructor provides a uniform representation for these structures,

$$\begin{array}{lcl} MdHeader & = & Bool \\ Md\ \ \tau & = & (MdHeader, \tau), \end{array}$$

and the function $valid(d)$ extracts the boolean value, returning $True$ if there are no errors in the structure, and $False$ otherwise. Each of the the *load* functions generated from specifications maintain the invariant that the the boolean value at each level of structure is $True$ if and only if all of the nested values are also $True$. Thus, one can test for errors during loading simply by examining the top-level boolean value. Note that it would be simple to augment this structure with additional information, such as the number of errors, an optional error message, or file system attributes, as in Figure 3; one would simply have to change the $MdHeader$ type and the $valid(d)$ function.

Load Functions. The most common use of Forest is as a tool for loading data from the file system into an in-memory representation. The functions defined by the inference rules in Figure 10 (a) implement this task. Formally, the judgment $\mathcal{E}; r; s \vdash \mathsf{load}\ F\ \rhd\ (v, d)$ holds if loading the specification s from the file system F at path r in environment $\mathcal{E}$ yields a pair (v, d) of representation and metadata. This judgment may be seen as describing a total function from $\mathcal{E}$, r, s and F to (v, d). The fact that the load function is total is useful—it allows the programmer to explore fragments of the file system that do not match s exactly. When F does not match s, the function puts default values in the representation v and records an error in the metadata d.

Let us examine the inference rules that define the load functions in detail. We assume that each constant $k^{\tau_2}_{\tau_1}$ has an associated $load_k$ function. For example, the *load* function for the File construct, which describes any file (but not symbolic links or directories), takes an environment $\mathcal{E}$, a file system F, and a path r as arguments, and either returns the contents and attributes of the file at path r, or " " and default attributes if $F(r)$ is undefined or not a file.

$$load_{\mathsf{File}}\ (\mathcal{E}, F, r) = \begin{cases} (u, (True, a)), \text{if} F(r) = (a, \mathsf{File}\ u) \\ (\text{“ ”}, (False, a_{\mathsf{default}})), \text{otherwise} \end{cases}$$

The *load* function for symbolic links is similar:

$$load_{\mathsf{Link}}\ (\mathcal{E}, F, r) = \begin{cases} (r', (True, a)), \text{if} F(r) = (a, \mathsf{Link}\ r') \\ (\cdot, (False, a_{\mathsf{default}})), \text{otherwise} \end{cases}$$

Note that this function only returns the path r' contained in the link and not the directory or file pointed to by it. To access the contents of the file system at r', the programmer could bind the path value returned by Link to a variable, and use the path construct to navigate to that path, as in the following specification:

$$\langle x : \mathsf{Link}\ , (x{::}\mathsf{File})\ \rangle$$

Alternatively, one could design another constant for symbolic links that implements a "deep" lookup in the file system.

The load function for path specifications $e{::}s$ evaluates the expression r/e to a path r', and invokes the load function for s from r'. The load function for dependent pairs $\langle x{:}s_1, s_2\rangle$ first invokes the load function for s_1, yielding a representation v_1 and metadata d_1, and then invokes the load function for s_2 in an extended environment where x is bound to v_1 and x_{md} to d_1, yielding v_2 and d_2. It returns (v_1, v_2) and $(valid(d_1) \wedge valid(d_2), (d_1, d_2))$ as the result. The load function for comprehensions $[s \mid x \in e]$ is similar. It evaluates the expression e to a list $[w_1, .., w_n]$, invokes the load function for s n times, in a series of environments with x bound to each w_i, and collects up the results into lists of representations $[v_1, .., v_n]$ and metadata $[d_1, .., d_n]$.

The load function for predicates $\mathsf{P}(e)$ tests whether the expression e is holds in the environment $\mathcal{E}$. It returns $()$ as the representation and $(b, ())$ as the metadata, where b is $\llbracket e \rrbracket^{\mathcal{E}}_{bool}$. The final load function, for options, $s?$ is defined using two inference rules. The first handles the case where the current path r exists in the file system. It uses s's load function to obtain a representation and metadata and injects both into the $Maybe$ type using the Just constructor. The second handles the case where r does not exist in the file system. It returns $Nothing$ for both the representation and metadata.

Store Functions. Just as they can be used for loading from the file system, Forest specifications can also be used to store data back to the file system. The inference rules in Figure 10 (b) define the judgment $\mathcal{E}; r; s \vdash \mathsf{store}\ (F, v, d)\ \rhd\ (F', \phi')$, which holds if storing (v, d) into F at r using specification s with respect to $\mathcal{E}$ yields a modified file system F' and a *validator* ϕ'. This last element, the validator, is a predicate on file systems that tests for inconsistencies during storing — it will be described in greater detail momentarily. As with load functions, the store judgment may be read as a total function; it maps $\mathcal{E}$, r, s, F, and (v, d) to (F', ϕ').

Intuitively, it should be obvious that there are numerous ways that storing data back to the file system could go wrong. For example, the representation v and metadata d might not be consistent with the existing information on the file system, or with each other. Even worse, storing v and d back to F could involve overwriting the same file (or link, or directory) with multiple pieces of data—a conflict. Thus, to reliably use store functions to manipulate the file system, we need a way way to track the intricate constraints on the data in the representation, metadata, and file system implied by the specification.

The validator ϕ' produced by the store function does exactly this—it keeps track of the conditions needed to ensure that storing the representation and metadata back to F will accurately reflect all of the information they contain. As a simple example that shows why validators are necessary, consider a specification s defined as $\langle x{:}\mathsf{File}, \mathsf{File}\rangle$ and suppose that we invoke s's load function with $\mathcal{E}$, F, and r where $F(r)$ is $(a, \mathsf{File}\ u)$. According to the semantics of the load functions described above, the representation v will be a pair (u, u) containing two copies of the file contents at r and the metadata d will also be a pair $((True, a), (True, a))$ containing two copies of the attributes associated to that file. Now suppose that we change the representation to (u, u'), where $u \neq u'$, and store the result back to the file system. Because this new representation does not satisfy the dependency between the two components of the pair implied by s, the store function cannot produce a new file system containing the information in both u and u'. It must store one of the strings and discard the other. The validator ϕ' detects this inconsistency. In this case, the validator will be equivalent to the following predicate on file systems:

$$\lambda F'.\ (F'(r) = (a, \mathsf{File}\ u)) \wedge (F'(r) = (a, \mathsf{File}\ u'),$$

which is clearly not satisfiable unless $u = u'$.

The store functions also use validators to detect internal inconsistencies between the representation and metadata. For example, consider the specification s defined as $s_1?$, and suppose that invoking s's load function with $\mathcal{E}$, F, r where $F(r)$ is undefined. The representation and metadata will be $\mathit{Nothing}$ and $(\mathit{True}, \mathit{Nothing})$ respectively. Now suppose that we change the representation to $\mathit{Just}\ u'$, and invoke s's store function. Even though the pair $(\mathit{Just}\ u', (\mathit{True}, \mathit{Nothing}))$ would never be produced by s's load function, the store function must still do something reasonable with it. A reasonable choice—the one used in our semantics—is to update the file system contents by passing $(\mathsf{File}\ u', a_{\mathsf{default}})$ to s's store function, but produce a validator $\phi' = \lambda F'.\ \mathit{False}$ that records the inconsistency between the representation and metadata. This informs the programmer that if they invoke the load function on the new file system, the representation and metadata may not be preserved—*e.g.*, loading using s_1 could result in an error.

Now let us examine the inference rules in detail. We assume that each constant defines a store_k function. The store function for File is defined as follows:

$$\mathit{store}_{\mathsf{File}}\ (\mathcal{E}, F, r, v, d) =$$
$$\begin{cases} (F[r := (a, \mathsf{File}\ v)], \\ \ \lambda F'.\ (F'(r) = (a, \mathsf{File}\ v))) \\ \qquad\qquad\qquad\qquad\qquad \text{if } d = (\mathit{True}, a) \\ (F[r := \bot], \\ \ \lambda F'.\ (v = \text{``''} \wedge a = a_{\mathsf{default}} \wedge F'(r) \neq (\_, \mathsf{File}\ \_))) \\ \qquad\qquad \text{if } d = (\mathit{False}, a)\ \wedge F(r) = (\_, \mathsf{File}\ \_) \\ (F, \\ \ \lambda F'.\ (v = \text{``''} \wedge a = a_{\mathsf{default}} \wedge F'(r) \neq (\_, \mathsf{File}\ \_))) \\ \qquad\qquad \text{if } d = (\mathit{False}, a)\ \wedge F(r) \neq (\_, \mathsf{File}\ \_) \end{cases}$$

It overwrites the contents of the file system F at path r with $(a, \mathsf{File}\ v)$ if d is valid, deletes the contents of F at r if d is not valid but $F(r)$ contains a file, and returns F unchanged otherwise. The validator ϕ' also has three cases: in the first, it tests whether $F'(r)$ is $(a, \mathsf{File}\ v)$; in the second and third cases it tests whether $F'(r)$ is not a file and (v, a) is the default (“”, a_{default}) generated by the load function. These constraints are necessary to prove the round-tripping properties described at the end of this section.

The rule for path specifications $e{::}s$ simply passes off control to the store function for s after replacing the current path r with $\llbracket r\,/\,e \rrbracket^{\mathcal{E}}_{path}$. The rule for dependent pairs $\langle x{:}s_1, s_2\rangle$ is more interesting. Given a pair (v_1, v_2) as the representation and $(b, (d_1, d_2))$ as the metadata, it invokes the store function for s_1 on (v_1, d_1), yielding an updated file system F_1' and validator ϕ_1', and then invokes the store function for s_2 on (v_2, d_2) in an extended environment where x is bound to v_1 and x_{md} is bound to d_1, yielding another updated file system F_2' and validator ϕ_2'. It combines the updated file systems using the right-biased file system concatenation operator defined as follows:

$$(F_1{+\!+}F_2)(r) =$$
$$\begin{cases} (a_2, \mathsf{Dir}\ (U_1 \cup U_2)) & \text{if } F_1(r) = (a_1, \mathsf{Dir}\ U_1)\ \wedge \\ & \quad F_2(r) = (a_2, \mathsf{Dir}\ U_2) \\ F_1(r) & \text{if } F_2(r) = \bot \\ F_2(r) & \text{otherwise} \end{cases}$$

Finally, it conjoins the two validators ϕ_1' and ϕ_2'. The result is a file system that contains the consistent changes made to the file system by the store functions for s_1 and s_2 as well as a validator that checks for the consistency of all the changes. The store function for comprehensions is similar.

The store function for predicates $\mathsf{P}(e)$ returns the input file system F unchanged and a validator ϕ' that checks whether e evaluates to b, the boolean value stored in the metadata. The store function for options $s_1?$ is defined by three inference rules. The first handles the case where the representation is $\mathit{Just}\ v_1$ and the metadata is $\mathit{Just}\ d_1$ by simply unpacking the encapsulated values and invoking s_1's store function. The second handles the case where the representation is $\mathit{Nothing}$. It deletes the file and returns a validator that checks whether the metadata d is also $\mathit{Nothing}$ and the file system is undefined on the path r. The third rule handles the case where the representation is $\mathit{Just}\ v_1$ and the metadata is $\mathit{Nothing}$. It invokes s_1's store function on v_1 and default metadata and returns a validator that always evaluates to False, reflecting the inconsistency in the representation and metadata.

s	$\mathcal{R}\llbracket s \rrbracket$	$\mathcal{M}\llbracket s \rrbracket$
$k^{\tau_2}_{\tau_1}$	τ_1	$\mathit{Md}\ \tau_2$
$e{::}s$	$\mathcal{R}\llbracket s \rrbracket$	$\mathcal{M}\llbracket s \rrbracket$
$\langle x{:}s_1, s_2\rangle$	$(\mathcal{R}\llbracket s_1 \rrbracket, \mathcal{R}\llbracket s_2 \rrbracket)$	$\mathit{Md}\ (\mathcal{M}\llbracket s_1 \rrbracket, \mathcal{M}\llbracket s_2 \rrbracket)$
$[s \mid x \in e]$	$[\mathcal{R}\llbracket s \rrbracket]$	$\mathit{Md}\ [\mathcal{M}\llbracket s \rrbracket]$
$\mathsf{P}(e)$	$()$	$\mathit{Md}\ ()$
$s?$	$\mathit{Maybe}\ \mathcal{R}\llbracket s \rrbracket$	$\mathit{Md}\ (\mathit{Maybe}\ \mathcal{M}\llbracket s \rrbracket)$

Figure 9. Forest calculus representation and metadata types.

7.4 Formal Properties

The semantics of Forest calculus specifications is carefully designed to ensure some essential correctness properties. The first is a basic type safety property, which states that the load function for specifications s generates representations and metadata belonging to $\mathcal{R}\llbracket s \rrbracket$ and $\mathcal{M}\llbracket s \rrbracket$ respectively.

Proposition 1 (Load Type Safety)
If $\mathcal{E}; r; s \vdash \mathsf{load}\ F\ \rhd\ (v, d)$ and $\mathcal{R}\llbracket s \rrbracket = \tau_{\mathcal{R}}$ and $\mathcal{M}\llbracket s \rrbracket = \tau_{\mathcal{M}}$ then $\vdash v : \tau_{\mathcal{R}}$ and $\vdash d : \tau_{\mathcal{M}}$,

This property demonstrates that our type definitions are properly aligned with the semantics of loading.

To ensure that the semantics of loading is aligned with the semantics of storing, we also prove the following pair of round-tripping properties.

Theorem 2 (LoadStore)
Let $\mathcal{E}$ be an environment, F a file system, r a path, s a specification, v a representation, and d metadata. If

$$\begin{gathered} \mathcal{E}; r; s \vdash \mathsf{load}\ F\ \rhd\ (v, d) \\ \mathcal{E}; r; s \vdash \mathsf{store}\ (F, v, d)\ \rhd\ (F', \phi) \end{gathered}$$

then $F = F'$ and $\phi'(F')$.

Theorem 3 (StoreLoad)
Let $\mathcal{E}$ be an environment, F and F' file systems, r a path, s a specification, v a representation, d and d' metadata, and ϕ' a validator. If

$$\begin{gathered} \mathcal{E}; r; s \vdash \mathsf{store}\ (F, v, d)\ \rhd\ (F', \phi') \qquad \phi'(F') \\ \mathcal{E}; r; s \vdash \mathsf{load}\ F'\ \rhd\ (v', d') \end{gathered}$$

then $(v', d') = (v, d)$.

The first theorem states that loading from a file system F and immediately storing the resulting representation and metadata yields the original file system and, moreover, that this file system will satisfy the validator produced by the store function. This guarantees that the store function will not disturb information in the file system if possible, such as the information outside of the fragment of the

$$\boxed{\mathcal{E}; r; s \vdash \textsf{load}\ F\ \rhd\ (v, d)}$$

$$\frac{}{\mathcal{E}; r; k_{\tau_1}^{\tau_2} \vdash \textsf{load}\ F\ \rhd\ (\mathit{load}_k(\mathcal{E}, F, r))}$$

$$\frac{\mathcal{E}; [\![r\,/\,e]\!]_{Path}^{\mathcal{E}}; s \vdash \textsf{load}\ F\ \rhd\ (v, d)}{\mathcal{E}; r; e{::}s \vdash \textsf{load}\ F\ \rhd\ (v, d)}$$

$$\frac{\begin{array}{c}\mathcal{E}; r; s_1 \vdash \textsf{load}\ F\ \rhd\ (v_1, d_1) \\ (\mathcal{E}, x \mapsto v_1, x_{md} \mapsto d_1); r; s_2 \vdash \textsf{load}\ F\ \rhd\ (v_2, d_2) \\ b = \mathit{valid}(d_1) \wedge \mathit{valid}(d_2)\end{array}}{\mathcal{E}; r; \langle x{:}s_1, s_2\rangle \vdash \textsf{load}\ F\ \rhd\ ((v_1, v_2), (b, (d_1, d_2)))}$$

$$\frac{\begin{array}{c}[\![e]\!]_{[\tau]}^{\mathcal{E}} = [w_1, .., w_k] \\ \forall i \in \{1, .., k\}.\ (\mathcal{E}, x \mapsto w_i); r; s \vdash \textsf{load}\ F\ \rhd\ (v_i, d_i) \\ b = \bigwedge_i^k \mathit{valid}(d_i) \quad vs = [v_1, .., v_k] \quad ds = [d_1, .., d_k]\end{array}}{\mathcal{E}; r; [s \mid x \in e] \vdash \textsf{load}\ F\ \rhd\ (vs, (b, ds))}$$

$$\frac{b = [\![e]\!]_{Bool}^{\mathcal{E}}}{\mathcal{E}; r; \mathsf{P}(e) \vdash \textsf{load}\ F\ \rhd\ ((), (b, ()))}$$

$$\frac{r \notin \textsf{dom}(F)}{\mathcal{E}; r; s_1? \vdash \textsf{load}\ F\ \rhd\ (\mathit{Nothing}, (\mathit{True}, \mathit{Nothing}))}$$

$$\frac{r \in \textsf{dom}(F) \quad \mathcal{E}; r; s_1 \vdash \textsf{load}\ F\ \rhd\ (v_1, d_1)}{\mathcal{E}; r; s_1? \vdash \textsf{load}\ F\ \rhd\ (\mathit{Just}\ v_1, (\mathit{valid}(d_1), \mathit{Just}\ d_1))}$$

(a)

$$\boxed{\mathcal{E}; r; s \vdash \textsf{store}\ (F, v, d)\ \rhd\ (F', \phi')}$$

$$\frac{}{\mathcal{E}; r; k_{\tau_1}^{\tau_2} \vdash \textsf{store}\ (F, v, d)\ \rhd\ (\mathit{store}_k(\mathcal{E}, F, r, v, d))}$$

$$\frac{\mathcal{E}; [\![r\,/\,e]\!]_{Path}^{\mathcal{E}}; s \vdash \textsf{store}\ (F, v, d)\ \rhd\ (F', \phi')}{\mathcal{E}; r; e{::}s \vdash \textsf{store}\ (F, v, d)\ \rhd\ (F', \phi')}$$

$$\frac{\begin{array}{c}\mathcal{E}; r; s_1 \vdash \textsf{store}\ (F, v_1, d_1)\ \rhd\ (F_1', \phi_1') \\ (\mathcal{E}, x \mapsto v_1, x_{md} \mapsto d_1); r; s_2 \vdash \textsf{store}\ (F, v_2, d_2)\ \rhd\ (F_2', \phi_2') \\ \phi' = \lambda F'.\ (b = \mathit{valid}(d_1) \wedge \mathit{valid}(d_2)) \wedge \phi_1'(F') \wedge \phi_2'(F')\end{array}}{\mathcal{E}; r; \langle x{:}s_1, s_2\rangle \vdash \textsf{store}\ (F, (v_1, v_2), (b, (d_1, d_2)))\ \rhd\ (F_1'{+\!+}F_2', \phi')}$$

$$\frac{\begin{array}{c}vs = [v_1, .., v_j] \quad ds = [d_1, .., d_l] \\ [\![e]\!]_{[\tau]}^{\mathcal{E}} = [w_1, .., w_m] \quad k = \min(j, l, m) \\ \forall i \in \{1, .., k\}.\ (\mathcal{E}, x \mapsto w_i); r; s \vdash \textsf{store}\ (F, v_i, d_i)\ \rhd\ (F_i', \phi_i') \\ \phi' = \lambda F'.\ (j = l = m) \wedge (b = \bigwedge_i^k \mathit{valid}(d_i)) \wedge (\bigwedge_i^k \phi_i'(F'))\end{array}}{\mathcal{E}; r; [s \mid x \in e] \vdash \textsf{store}\ (F, vs, (b, ds))\ \rhd\ (F_1'{+\!+}..{+\!+}F_k', \phi')}$$

$$\frac{\phi' = \lambda F'.\ (b = [\![e]\!]_{Bool}^{\mathcal{E}})}{\mathcal{E}; r; \mathsf{P}(e) \vdash \textsf{store}\ (F, (), (b, ()))\ \rhd\ (F, \phi')}$$

$$\frac{\begin{array}{c}\mathcal{E}; r; s_1 \vdash \textsf{store}\ (F, v_1, d_1)\ \rhd\ (F', \phi_1') \\ \phi' = \lambda F'.\ (b = \mathit{valid}(d_1)) \wedge (r \in \textsf{dom}(F')) \wedge \phi_1'(F')\end{array}}{\mathcal{E}; r; s_1? \vdash \textsf{store}\ (F, \mathit{Just}\ v_1, (b, \mathit{Just}\ d_1))\ \rhd\ (F', \phi')}$$

$$\frac{\phi' = \lambda F'.\ (d = \mathit{Nothing}) \wedge b \wedge r \notin \textsf{dom}(F')}{\mathcal{E}; r; s_1? \vdash \textsf{store}\ (F, \mathit{Nothing}, (b, d))\ \rhd\ (F[r := \bot], \phi')}$$

$$\frac{\begin{array}{c}\mathcal{E}; r; s_1 \vdash \textsf{store}\ (F, v_1, d_{\textsf{default}}^{s_1})\ \rhd\ (F', \phi_1') \\ \phi' = \lambda F'.\ \mathit{False}\end{array}}{\mathcal{E}; r; s_1? \vdash \textsf{store}\ (F, \mathit{Just}\ v_1, (b, \mathit{Nothing}))\ \rhd\ (F', \phi')}$$

(b)

Figure 10. Forest calculus semantics for (a) loading and (b) storing.

file system described by the specification. It also establishes that the validator is not the trivial predicate on file systems that always returns false.

The second theorem states that storing an arbitrary representation and then loading the resulting file system yields the same representation and metadata, provided the stored file system satisfies the validator. This theorem ensures that the store function reflects all of the information contained in the representation in the updated file system.

These properties are based on the general correctness conditions that have been proposed for bidirectional transformations in the context of lenses [10], but are generalized here to accommodate the inconsistencies that can arise when working with imperfect, ad hoc data. The proofs of these theorems can be found in the accompanying technical report.

8. Related Work

The work in this paper builds upon ideas developed in the Pads project [5, 7]. Pads uses extended type declarations to describe the grammar of a document and simultaneously to generate types for parsed data and a suite of data-processing tools. The obvious difference between Pads (and other parser generators) and Forest is that Pads generates infrastructure for processing strings (the insides of a single file) whereas Forest generates infrastructure for processing entire filestores. In addition, Forest (and Pads/Haskell) is architecturally superior to previous versions of Pads in the tight integration with its host language and in its support for third-party generic programming and tool construction.

More generally, Forest shares high-level goals with other systems that seek to make data-oriented programming simpler and more productive. For example, Microsoft's LINQ [20] extends the .NET languages to enable querying any data source that supports the `IEnumerable` interface using a simple, convenient syntax. LINQ differs from Forest in that it does not provide support for declaratively specifying the structure of, and then ingesting, filestores. As a second example, *Type Providers* [28], an experimental feature of F#, help programmers materialize standard data sources equipped with predefined schemas (such as XML documents or databases) in memory in an F# program. Type Providers do not themselves provide a new means for describing data sources (as Forest does).

Several XML-based languages for specifying file formats, file organization and file locations have been proposed. One example of such a language is XFiles [1]. XFiles uses RDF specifications to describe the location, permissions, ownership, and other attributes of files, as well as the name of an application capable of parsing specific files. The key difference between XFiles and Forest is that Forest is tightly integrated into a general-purpose, conventional programming language. Forest declarations generate types, functions and data structures that materialize the data within a surrounding Haskell program while XFiles does not interoperate directly with a conventional programming language.

A recent MSc thesis by Ntzik proposes using an extension of context logic [2] to reason about the effects of updates made to file systems using standard POSIX commands [24]. The core goal of Ntzik's work is to create a new kind of Hoare Logic, and consequently, it is quite different from Forest.

The round-tripping properties that core Forest programs obey are based on laws that have been proposed in the context of well-behaved bidirectional transformations, often called lenses [10]. As far as we are aware, lenses for file systems have not been developed but some of the same fundamental issues that arise in core Forest have been studied by Hu and his colleagues, including languages that handle data with internal dependencies [23] and ones that handle graph structures [17].

9. Conclusions

In this paper, we propose the idea of extending a modern, high-level programming language with tightly integrated features for processing coherent file system fragments, which we call filestores. To demonstrate the potential of this idea, we designed Forest, a domain-specific language embedded in Haskell for describing and managing filestores.

The Forest design has been informed by both theoretical analysis and practical experience. On the theoretical side, we developed a formal calculus that models the core Forest functionality and we proved that our calculus obeys round-tripping laws derived from previous work on bi-directional programming paradigms. On the practical side, we illustrated the utility of our design by describing several example filestores, and showing how to use these descriptions to build simple Haskell scripts that query, analyze, and transform the example data in useful ways. We also provided evidence that Forest has effective support for building generic, description-directed tools by implementing a number of such tools ourselves, including a filestore visualizer, a generic query interface, an access control checker, and (circularly) a simple description inference engine. An ancillary benefit of this engineering work is that it serves as an extensive case study in domain-specific language design, and, as such, inspired changes in the design of Template Haskell.

For further information about Forest, we direct readers to the Forest web site [9], where they may find our open source implementation and a number of additional examples.

Acknowledgments

We wish to thank Simon Peyton Jones for extending Haskell's quasiquoting mechanism to support the Forest design and for assisting us in its use, John Launchbury for helping us design and implement Pads/Haskell, and the anonymous ICFP reviewers for many insightful comments and suggestions.

This work was supported in part by the NSF under grant CCF-1016937, the ONR under grant N00014-09-1-0652, and the NSFC under grant 6103302. Any opinions, findings, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

References

[1] S.-C. Buraga. An XML-based semantic description of distributed file systems. In *RoEduNet*, pages 41–48, 2003.

[2] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL*, pages 271–282, 2005.

[3] Filesystem Hierarchy Standard Group. Filesystem hierarchy standard. `http://www.pathname.com/fhs/`, 2004.

[4] K. Fisher, N. Foster, D. Walker, and K. Q. Zhu. Forest 1.0: A Language and Toolkit for Programming with Filestores. Technical Report TR-904-11, Princeton University, June 2011.

[5] K. Fisher and R. Gruber. PADS: A domain specific language for processing ad hoc data. In *PLDI*, pages 295–304, June 2005.

[6] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *POPL*, Jan. 2006.

[7] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. *JACM*, 57:10:1–10:51, February 2010.

[8] K. Fisher and D. Walker. The PADS project: An overview. In *Proceedings of the 14th International Conference on Database Theory*, ICDT '11, pages 11–17, New York, NY, USA, 2011. ACM.

[9] Forest: A language and toolkit for programming with file system fragments. `http://forestproj.org`, 2010.

[10] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. *TOPLAS*, 29(3), May 2007.

[11] M. J. Freedman. Experiences with CoralCDN: A five-year operational view. In *NSDI*, pages 7–7, 2010.

[12] M. J. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with Coral. In *NSDI*, pages 18–18, 2004. See also `http://www.coralcdn.org/`.

[13] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30:1203–1233, September 2000.

[14] Haskell Graphviz Package. `http://hackage.haskell.org/package/graphviz`.

[15] Haskell Source Extensions Package. `http://hackage.haskell.org/package/haskell-src-exts`.

[16] Haskell Source Meta Package. `http://hackage.haskell.org/package/haskell-src-meta`.

[17] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *ICFP*, pages 205–216, 2010.

[18] R. Lämmel and S. P. Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *TLDI*, pages 26–37, 2003.

[19] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[20] LINQ: .NET language-integrated query. `http://msdn.microsoft.com/library/bb308959.aspx`, Feb. 2007.

[21] G. Mainland. Why it's nice to be quoted: Quasiquoting for Haskell. In *Haskell Workshop*, pages 73–82, 2007.

[22] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernández, and A. Gleyzer. PADS/ML: A functional data description language. In *POPL*, Jan. 2007.

[23] S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bidirectional updating. In *APLAS*, pages 2–20, Nov. 2004.

[24] G. Ntzik. Local reasoning for filesystems. Master's thesis, Imperial College, Sept. 2010.

[25] PADS project. `http://www.padsproj.org/`, 2007.

[26] T. J. Parr and R. W. Quong. ANTLR: A predicated–$LL(k)$ parser generator. *Softw. Pract. Exper.*, 25(7):789–810, July 1995.

[27] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Haskell Workshop*, pages 1–16, 2002.

[28] D. Syme. Looking Ahead with F#: Taming the Data Deluge. Presentation at the Workshop on F# in Education, Nov. 2010.

[29] Template Haskell Extension Proposal. `hackage.haskell.org/trac/ghc/blog/Template%20Haskell%20Proposal`.