Forest: A Language and Toolkit For Programming with Filestores

Abstract

Many applications use the file system as a simple persistent data store. This approach is expedient, but not robust. The correctness of such an application depends on the collection of files, directories, and symbolic links having a precise organization. Furthermore these components must have acceptable values for a variety of file system attributes such as ownership, permissions, and timestamps. Unfortunately, current programming languages do support documenting assumptions about the file system. In addition, actually loading data from disk requires writing tedious boilerplate code.

This paper describes Forest, a new domain-specific language embedded in Haskell for describing directory structures. Forest descriptions use a type-based metaphor to specify portions of the file system in a simple, declarative manner. Forest makes it easy to connect data on disk to an isomorphic representation in memory that can be manipulated by programmers as if it were any other data structure in their program. Forest generates metadata that describes to what degree the files on disk conform to the specification, making error detection easy. As a result, the system greatly lowers the divide between on-disk and in-memory representations of data. Forest leverages Haskell's powerful generic programming infrastructure to make it easy for third-party developers to build tools that work for any Forest description. We illustrate the use of this infrastructure to build a number of useful tools, including a visualizer, permission checker, and description-specific replacements for a number of standard shell tools. Forest has a formal semantics based on classical tree logics.

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 conventional files scattered across 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 [6, 7]. The logs are organized in hierarchical directory structures based on machine name, time and date. Freedman mines the logs for information on system security, health and performance. At Harvard, Vinothan Manoharan, a physics professor, stores his experimental data in sets of files and extracts information using python scripts. At AT&T, vast structured repositories contain networking information, phone call detail and billing data. And there are many other examples 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) possibly writing programs that transform raw data so it may be loaded; (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 paying 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 and file contents to organize the data. We will call such a representation of a coherent set of data a *filestore*. The "query language" for a filestore is often a shell script or conventional programming language.

Unfortunately, the informality of filestores can have 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, 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 data 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 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.

In this paper, we propose a better way: A novel type-based specification language, programming environment and toolkit for managing filestores. This language, called Forest, is an embedded domain-specific language in Haskell. Forest allows programmers to describe the expected shape of a filestore and to lazily materialize it as format-specific Haskell data structures with the goal of making programming with filestores as easy and robust as programming with any other Haskell data structure. Furthermore, Forest leverages Haskell's support for generic programming to make it easy to define tools that work for any filestore with a Forest description.

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 as described by the Filesystem Hierarchy Standard Group in an informal standards document [2]. In addition, the standard requires that certain directories contain *only* the specified files, presumably for security reasons. Forest provides a language for writing such standards precisely and a checker that allows users to verify that their installation conforms to the standard. As another example, the Pads website [17] 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.

In exchange for writing a Forest specification, programmers obtain a number of benefits. These benefits include (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, and (4) type class instance declarations that make it possible for programmers to query, analyze, and transform filestore data using generic functions.

To illustrate the power of the generic programming infrastructure and to increase the value of writing a Forest specification, we built a number of tools that work for any filestore with a Forest specification. These tools include a filestore visualizer, a permission checker, and filestore-specific versions of standard command-line tools such as `grep` and `tar`. Using this same infrastructure, we built a tool to infer a Forest specification for the directory structure starting at a given path, which means programmers do not have to start from scratch when writing a specification for their filestores.

In summary, this paper makes the following contributions.

- **Conceptual**: We propose the *idea* of extending a modern programming language with *tightly integrated linguistic features for describing filestores* and for automatically generating programming infrastructure from such descriptions.
- **Language Design**: We present the design of Forest and illustrate its use on real-world examples. The design is expressive, concise and smoothly integrated into Haskell. It is supported by a formal semantics inspired by classical tree logics.
- **Tool Generation**: We describe how third-party developers can use Haskell's generic programming infrastructure to generate filestore-specific tools generically.
- **Case Study in Domain-Specific Language Design**: Forest is fully implemented and its design serves as a case study in extensive, practical, domain-specific language design in modern languages by combining a number of experimental features of Haskell such as quasi-quoting and Template Haskell. Moreover, our Forest design and implementation experience has had practical impact on the Haskell implementation itself: the Haskell team modified and extended Template Haskell and the quasi-quoting mechanism in response to our needs. These modifications are available in the most recent release of Haskell.

2. Example Filestores

In this section, we present two example filestores. We will 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 in the filestore to decide on undergraduate awards and to track grading trends. As it has evolved over time, there have been a few slight changes in format – typical for ad hoc filestores. Naturally, any description needs to cope with the variation.

Figure 1 shows a snippet of the (anonymized) student filestore designed to illustrate its structure. At the top level, there are 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 and Engineering 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 such directory can also contain a template file named `sss.txt` (or something similar) for creating new students.

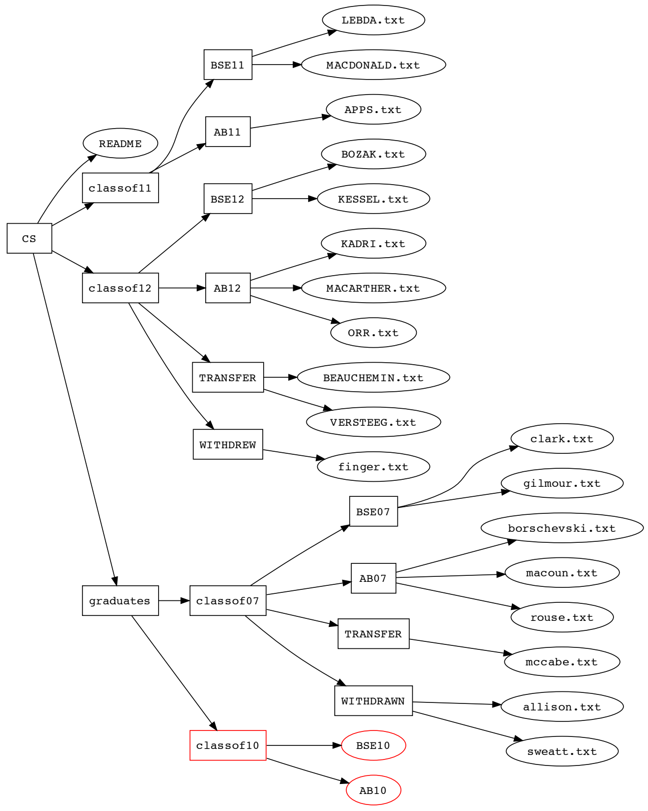

Figure 1. Anonymized snippet of Princeton computer science undergraduate data. The red denotes an error, in this case, missing files.

The second filestore contains log files for CoralCDN [6, 7]. To monitor the performance, health, 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 with a top-level `logs` directory containing a set of subdirectories, one for each host. Each such host directory contains another set of directories, labeled by date and time. Finally, each of these directories contains 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.

3. Forest Design

Forest is a domain-specific language embedded within Haskell using the Quasiquote mechanism [15]. In a typical Forest description, Forest declarations are interleaved with ordinary Haskell declarations. To introduce new Forest declarations, the programmer simply opens the Forest sublanguage scope:

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

Forest uses a type-based metaphor to describe directory structures, so once within the Forest sublanguage, the programmer writes declarations that look and feel very much like extended Haskell type declarations. Each such type declaration serves three purposes: (1) it describes a fragment of the file system, (2) it specifies the structure of the in-memory *representation* that will be constructed when the fragment is (lazily) loaded into a Haskell program, and (3) it specifies the structure of the in-memory *metadata* that will be generated when the fragment loaded. Such metadata

```
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 2. Forest metadata types.

includes error information (missing file, insufficient permissions, *etc.*) as well as file system attributes (owner, size, *etc.*). As we explain the design of Forest, readers should keep these three different aspects in mind. The effectiveness of the Forest language comes in part from the fact that these three elements can all be specified in a single compact description.

Every Forest description is defined relative to a *current path* within the file system. As Forest matches a description against the file system, it adjusts the current path to reflect its navigation.

At its core, Forest is a simple dependent type system in which base types denote files of various flavors and record types describe directories. Forest also includes a list type to describe collections of files that all share some type. We use a variety of other type constructors to build more refined structures from these basic building blocks. We discuss each of these constructs in turn in the remainder of this section. A note on terminology: we use the term "file system object" or more simply "object" to denote either a file, or a directory, or a symbolic link.

3.1 Files

Forest provides a small collection of base types for describing files: `Text` for ASCII files, `Binary` for binary files, and `Any` for arbitrary files. As with all Forest types, each of these types specifies a representation type, a metadata type, and a loading function. The in-memory representation for an ASCII file is a Haskell `String`; for binary and arbitrary files, it is a `ByteString`. For all three file types, the metadata type pairs file-system metadata with metadata describing properties of the file contents. The file-system metadata has type `Forest_md`, shown in Figure 2. This structure stores two kinds of information: (1) the number and kind of any errors that occurred while loading the file and (2) the attributes associated with the file. File-content metadata typically describes errors within the file, but can be used for other purposes. 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.

Of course, there are many kinds of files, and the appropriate representation and content metadata type for each such file varies. Possible examples include XML documents, Makefiles, source files in various languages, shell scripts, *etc.* To support such files, Forest provides a plug-in architecture, allowing third party users to define new file base types by specifying a representation type, a metadata type, and a corresponding loading function.

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 [5]. In such cases, Forest can leverage the Pads/Haskell data description language to define format-specific in-memory representations, content metadata, and loading functions. Pads/Haskell is a recently developed version of Pads [3, 4, 16]. 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) = < pads decl >
|]
```

This description is parameterized by the name of the student whose data is in the file; the complete description appears in the appendix. From this specification, the Pads compiler generates an in-memory representation type `Student`, a content metadata type `Student_md`, and a parsing function.

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.

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. Indeed, Pads/Haskell and Forest were designed 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, the rest of 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 Optional Files

Sometimes, a given file (or directory or symbolic link) may or may not be present in the file system, and either situation is valid. Forest provides the `Maybe` type constructor for this situation. If `T` is a Forest type, then `Maybe T` is the Forest type denoting an optional `T`. In particular, `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`.

3.3 Symbolic Links

When symbolic links occur in a described file system fragment, Forest follows the symbolic link to its target, mimicking standard shell behavior. In addition, however, it is possible for 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 such a specification in the next subsection.

3.4 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"     :: Text
  , 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.*, `Text` 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" :: Text
  , 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" :: Text
  , ... }
```

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. As we will see shortly, 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 neither is 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" :: Text
  , 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. This example also illustrates abstraction: any Forest declaration may be parameterized by specifying a legal Haskell expression identifier and its type. The types of the fields for `seniors` and `juniors` illustrate the use of parameterized descriptions. When an argument 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.

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.

3.5 List Comprehensions

Record directories allow programmers to specify a fixed number of file system objects, each with its own type. List comprehensions, 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 comprehension to specify the Grads directory from Figure 1:

```
> 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, comprehensions have the following form.

```
[path :: T | id <- gen, pred]
```

Here, 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 ids 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 comprehension is a 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 container type class that supports operations to convert between the list representation and the desired container representation.

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 Attributes and Constraints

Every file system object has a number of attributes associated with it, such as the file owner and size. In general, if a Forest identifier id refers to a path, the attributes for the object at that path are available through the identifier id_att, which has type Forest_md (Figure 2). Forest defines accessor functions such as get_modes (get permissions), get_kind (get ascii/binary/directory characteristics) and others to inspect these attributes.

```
[forest|
 data PrincetonCS (y::Integer) = Directory
   { notes   is "README" :: Text
   , 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 3. Forest description of Princeton filestore. (Associated Haskell and Pads code appears in auxiliary PLDI materials [5].)

Constrained types make use of attributes. For example, the type PrivateText specifies a text file accessible only by its owner.

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

The keyword **where** introduces a constraint on the underlying type. If a constraint is false, an error is registered in the metadata. Within the constraint, 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 :: Text
           | f <- matches (GL "*"),
          <| get_kind f_att == AsciiK |> ]
  , bin is [ b :: Binary
           | 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 "*"),
          <| get_sym s_att == True |> ]
  }
```

The description requires recursion to describe the directory case, which Forest supports. 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 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 if a binary file exists in the directory.

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

```
[forest|
  data Log = Directory
    { log is coralwebsrv :: Gzip (File CoralLog) }
  type Site  = [ d :: Log  | d <- matches time ]
  type Coral = [ s :: Site | s <- matches site ]
|]
```

Figure 4. Forest CoralCDN description. Associated Haskell and Pads code appears in auxiliary PLDI materials [5].

Load Functions:

```
log_load   :: FilePath -> IO (Log, Log_md),
site_load  :: FilePath -> IO (Site, Site_md)
coral_load :: FilePath -> IO (Top, Top_md),
```

Representation Types:

```
data     Log   = Log   {log :: CoralLog}
newtype Site  = Site  [(String, Log)]
newtype Coral = Coral [(String, Site)]
```

Metadata Types:

```
data Log_inner_md =
  Log_inner_md {log_md :: (Forest_md, CoralLog_md)}
type Log_md   = (Forest_md, Log_inner_md)
type Site_md  = (Forest_md, [(String, Log_md)])
type Coral_md = (Forest_md, [(String, Site_md)])
```

Figure 5. Coral load functions, representation and metadata types

3.7 Gzip and Tar Type Constructors

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 `CoralInfo` is a Pads/Haskell description of a CoralCDN log file then

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

describes a gzipped log file. Likewise, suppose `logs.tar.gz` is a gzipped tar file and that the type `Coral` 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 Coral = Gzip (Tar Coral)
```

3.8 Putting it all together

The previous subsections give an overview of the Forest design. Figures 3 and 4 show Forest specifications for our two running examples, minus the associated Pads/Haskell and Haskell declarations. The complete descriptions of these filestores and additional descriptions are available in the auxiliary PLDI materials [5], including descriptions of the Pads website, a Gene Ontology filestore, and the CVS repository structure.

4. Programming with Forest

Most 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.

To facilitate this style of programming, the Forest compiler generates several Haskell functions and types from every Forest description. It generates a *load function*, which traverses the file system and reads the files, directories, and symbolic links mentioned in the description into a structured object in memory; it generates a Haskell type for the in-memory *representation* of the data produced by the load function; and it generates a Haskell type for the *metadata* associated with the representation. For example, from the descriptions for CoralCDN logs in Figure 4, the compiler generates the load functions, the representation types, and the metadata types presented in Figure 5. 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.

As a simple example, consider the `Coral` description in Figure 4. The `coral_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) <- coral_load "/var/log/coral1"
```

Because `Coral` is a comprehension, both `rep` and `md` are lists. 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 records aggregate information about any errors encountered during loading as well as the file system attributes of each file, directory, or symbolic link loaded from the file system:

```
Forest_md
  { numErrors = 0,
    errorMsg = Nothing,
    fileInfo = FileInfo
      { fullpath = /var/log/coral,
        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 Coral l -> List.length l
```

More interestingly, since the internals of the web log are specified using Pads/Haskell (see auxiliary PLDI materials [5] 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 (Coral rs, (_,ms)) ->
    map (get_site *** get_mod) (zip rs ms)
```

These simple examples show how 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 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 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 [6]. 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_in e ]
```

Reading this program inside-out, we see that it first uses a list comprehension to iterate through `rep`, collecting the individual log entries in the `coralwebsrv.log.gz` file for incoming requests and projecting 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.

Overall, the main take-away from this section is how the Forest-generated infrastructure and tight coupling to Haskell facilitates construction of remarkably terse queries over the combination of file contents, file attributes and directory structure. Exploratory data analysis in this new programming paradigm is light-weight, easy and highly effective.

5. Tools

Third-party developers can use generic programming [12] to generate tools that will work for any file system structure that has a Forest description. 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 graph in Figure 1 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 red 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 [8, 9] 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 layout 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 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 Ptext,
    vERSTEEGtxt is "VERSTEEG.txt" :: File Ptext
}
...
```

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 comprehensions 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

We have fully implemented the Forest language and anticipate making it publicly available within the next few months. Haskell provides powerful language features and libraries that greatly facilitated the implementation of Forest. The most obvious of these features is the quasi-quoting [15] mechanism that we used to embed Forest into Haskell. This mechanism allowed us to enjoy the benefits of being an embedded domain-specific language without having to sacrifice the flexibility of defining our own syntax. To use quasi-quoting, we defined a Haskell value `forest` of type `QuasiQuoter` which specifies how to convert an input string representing a Forest declaration into the Template Haskell [18] data structures that represent the syntax of the corresponding collection of Haskell declarations. The quasi-quoting 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` quasi-quoter, 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 quasi-quote. Early versions of quasi-quoting supported quoting only expression and pattern forms. Simon Peyton Jones extended the mechanism to permit declaration and type quasi-quoting partly to enable the Forest implementation. We used this same approach to implement Pads/Haskell, which we built concomitantly.

$$
\begin{array}{rl}
\textit{Strings} & n \quad \in \Sigma^* \\
\textit{Paths} & r, s ::= \bullet \mid r \,/\, n \\
\textit{Attributes} & \mathsf{a} ::= \ldots \\
\textit{Filesystem} & T ::= \mathsf{File}(n) \\
\textit{Contents} & \qquad \mid \mathsf{Dir}(\{n_1, \ldots, n_k\}) \\
 & \qquad \mid \mathsf{Link}(r) \\
\textit{Filesystems} & F ::= \{\!|\, r_1 \mapsto (\mathsf{a}_1, T_1), \ldots r_k \mapsto (\mathsf{a}_k, T_k) \,|\!\} \\
\textit{Values} & v ::= \mathsf{a} \mid n \mid r \mid \mathsf{True} \mid \mathsf{False} \mid () \mid (v_1, v_2) \\
 & \qquad \mid \mathsf{Just(v)} \mid \mathsf{Nothing} \mid \{v_1, \ldots, v_k\} \\
\textit{Expressions} & e ::= x \mid v \mid \ldots \\
\textit{Environments} & \mathcal{E} ::= \bullet \mid E, x \mapsto v \\
\textit{Specifications} & s ::= k_{\tau_r}^{\tau_m} \mid \mathsf{Adhoc}(b_{\tau_r}^{\tau_m}) \mid e :: s \mid \langle x{:}s_1, s_2 \rangle \\
 & \qquad \mid \{s \mid x \in e\} \mid \mathsf{Pred}(e) \mid s?
\end{array}
$$

Figure 6. File systems and their specifications

Parsing. We used the parsec 3.1.0 parser combinator library [13] 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 [10] 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 [11], that provides 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 [20] 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 will be able to type check Forest sources directly.

7. A Core Calculus for Forest

We have defined an idealized core calculus that captures the essence of Forest. This calculus helped us to design various aspects of the language and provides a compact way of describing the central features of the language in a precise way. It is inspired by classical (*i.e.*, not separating, substructural or ambient) unordered tree logics, customized slightly to our application domain.

File system model and specification syntax. Figure 6 presents the formal file system model. File paths r are sequences of string names[1] and file systems F are finite partial maps from paths to pairs of file attributes a and file system contents T. We leave the attribute records abstract; they should include the usual fields: owner, group, date modified, *etc.* We write $\mathsf{a}_{\mathsf{default}}$ for a default attribute record where necessary. The contents T of a node in the file system may be a file $\mathsf{File}(n)$ (with underlying string contents n), a directory $\mathsf{Dir}(\{n_1, \ldots, n_k\})$ (with contents named $n_1, \ldots, n_k$) or a symbolic link $\mathsf{Link}(r)$ (where r is the path pointed to by the link).

A file system model F is *well-formed* if it is tree-shaped, with directories forming internal nodes and files and symbolic links at the leaves. In addition, these conditions must hold:

- The domain of F must be prefix-closed.
- If $F(r) = (\mathsf{a}, \mathsf{Dir}(\{n_1, \ldots, n_k\}))$ then for $i = 1, \ldots, k$, $r \,/\, n_i \in \mathsf{dom}(F)$.
- If $F(r) = (\mathsf{a}, \mathsf{File}(n_r))$ or $(\mathsf{a}, \mathsf{Link}(r'))$ then there does not exist n such that $r \,/\, n \in \mathsf{dom}(F)$.

Figure 6 also presents the syntax of a simple computation language e and our file system specifications s. The computation language e contains values v, variables x, and other operators, which we leave unspecified. An environment $\mathcal{E}$ maps variables to values. The semantic function $\mathsf{eval}_\tau(\mathcal{E}, F, r, e)$ evaluates an expression e in an environment $\mathcal{E}$ and file system F with respect to a current path r, yielding a value v of type τ.

The simplest file system specifications are constants k, which range over basic specifications such as those for files (F), text files (T), binary files (B), or any file system contents at all (A).

Pads/Haskell specifications are modeled as $\mathsf{Adhoc}(b^{\tau_m}_{\tau_r})$ where $b^{\tau_m}_{\tau_r}$ is a parser—i.e., a total function from pairs of environments and strings to pairs of type $\tau_r \times \tau_m$, where the first element is the representation for the parsed data and the second element is its metadata.

Forest's surface syntax combines specifications for records and paths into a single construct (and similarly for comprehensions and paths). The calculus models (dependent) records, paths, and comprehensions as independent, orthogonal constructs. Record specifications are written $\langle x{:}s_1, s_2\rangle$, where x may appear in s_2. Path specifications are written $e :: s$, where e is a path name (to be appended to the current path) and s specifies a fragment of the file system at that path. Comprehension specifications are written $\{s \mid x \in e\}$, where e is a set of values, x is a variable, and s, which may depend on x, specifies a fragment of the file system for each value of x. Forest's combined record-and-path construct `{c is "c.txt" :: C, d is "d.txt" :: D c}` is encoded in the calculus as $\langle x{:}(\texttt{"c.txt"} :: C), (\texttt{"d.txt"} :: D\ \ x)\rangle$. Similarly, Forest's comprehension `[x :: s | x <- e]` is encoded as the composition of the calculus constructors $\{s_1 \mid x \in e\}$ and $s_1 = x :: s$.

Predicate specifications $\mathsf{Pred}(e)$ succeed when e evaluates to True and fail when e evaluates to False under the current environment. A Forest constraint of the form `s where e` is encoded in the calculus as a dependent pair with a predicate: $\langle x{:}s, \mathsf{Pred}(e[x/\texttt{this}])\rangle$

Finally, maybe specifications are written as $s?$ in the calculus.

Calculus Semantics. The semantics of the calculus is organized into three separate definitions, one for each of the three artifacts generated by the Forest compiler. These definitions are spelled out in Figures 7 and 8.

[1] For simplicity, we ignore the special path elements ".." and ".". It is easy to add these features, although the semantics becomes more complicated because path expressions must be normalized.

$$\frac{}{\mathcal{E}; F; r \models k^{\tau_m}_{\tau_r} \rightsquigarrow ck(k^{\tau_m}_{\tau_r}, F, r)}$$

$$\frac{F(r) = (\mathsf{a}, \mathsf{File}(n)) \qquad b^{\tau_m}_{\tau_r}(E, n) = v, d}{\mathcal{E}; F; r \models \mathsf{Adhoc}(b^{\tau_m}_{\tau_r}) \rightsquigarrow v, (valid(d), (d, \mathsf{a}))}$$

$$\frac{F(r) = (\mathsf{a}, T) \qquad T \neq \mathsf{File}(n) \qquad b^{\tau_m}_{\tau_r}(E, \epsilon) = (v, d)}{\mathcal{E}; F; r \models \mathsf{Adhoc}(b^{\tau_m}_{\tau_r}) \rightsquigarrow v, (\mathsf{False}, (d, \mathsf{a}))}$$

$$\frac{r \notin \mathsf{dom}(F) \qquad b(E, \epsilon) = (v, d)}{\mathcal{E}; F; r \models \mathsf{Adhoc}(b^{\tau_m}_{\tau_r}) \rightsquigarrow v, (\mathsf{False}, (d, \mathsf{a}_{\mathsf{default}}))}$$

$$\frac{\mathcal{E}; F; \mathsf{eval}_{path}(E, F, r, r \,/\, e) \models s \rightsquigarrow v, d}{\mathcal{E}; F; r \models e :: s \rightsquigarrow v, d}$$

$$\frac{\begin{array}{c}\mathcal{E}; F; r \models s_1 \rightsquigarrow v_1, d_1 \\ \mathcal{E}[x \mapsto v_1, x_d \mapsto d_1]; F; r \models s_2 \rightsquigarrow v_2, d_2\end{array}}{\mathcal{E}; F; r \models \langle x{:}s_1, s_2\rangle \rightsquigarrow (v_1, v_2), (valid(d_1) \wedge valid(d_2), (d_1, d_2))}$$

$$\frac{\begin{array}{c}\mathsf{eval}_{(\tau\ set)}(\mathcal{E}, F, r, e) = \{v_1, \ldots, v_k\} \\ S = \{(v, d) \mid v' \in \{v_1, \ldots, v_k\} \text{ and } \mathcal{E}[x \mapsto v']; F; r \models s \rightsquigarrow v, d\}\end{array}}{\mathcal{E}; F; r \models \{s \mid x \in e\} \rightsquigarrow \pi_1\ S, (\bigwedge\ valid(\pi_2\ S), \pi_2\ S)}$$

$$\frac{}{\mathcal{E}; F; r \models \mathsf{Pred}(e) \rightsquigarrow (), (\mathsf{eval}_{bool}(E, F, r, e), ())}$$

$$\frac{r \notin \mathsf{dom}(F)}{\mathcal{E}; F; r \models s? \rightsquigarrow \mathsf{Nothing}, (\mathsf{False}, \mathsf{Nothing})}$$

$$\frac{r \in \mathsf{dom}(F) \qquad \mathcal{E}; F; r \models s \rightsquigarrow v, d}{\mathcal{E}; F; r \models s? \rightsquigarrow \mathsf{Just(v)}, (valid(d), \mathsf{Just(d)})}$$

Figure 7. Forest calculus semantics

The first semantic judgement has the form $\mathcal{E}; F; r \models s \rightsquigarrow v, d$. This judgement captures the behavior of the load function. Intuitively, it states that in environment $\mathcal{E}$ and file system F, specification s matches the file system fragment at current path r and produces the representation v and metadata d. This judgement may also be viewed as a total function from $\mathcal{E}$, F, r and s to the pair v and d. The judgement is total because when file system fragments fail to match the given specification, defaults are generated for the representation v and errors are recorded in the metadata d. This design is preferable to failing as it allows a programmer to explore a file system fragment even when it contains errors, as is common in filestores.

The rule for constants depends upon an auxiliary function *ck* (pronounced "check") that interprets the constants. For example, the *ck* function for the (F) construct, which describes any file (but not symbolic links or directories), is defined as follows:

$$\begin{array}{l} ck(\mathsf{F}^{att}_{string}, F, r) = (n, (\mathsf{True}, \mathsf{a})), \text{if} F(r) = (\mathsf{a}, \mathsf{File}(n)) \\ ck(\mathsf{F}^{att}_{string}, F, r) = (\texttt{""}, (\mathsf{False}, \mathsf{a})), \text{if} F(r) = (\mathsf{a}, T), T \neq \mathsf{File}(n) \\ ck(\mathsf{F}^{att}_{string}, F, r) = (\texttt{""}, (\mathsf{False}, \mathsf{a}_{\mathsf{default}})), \text{otherwise} \end{array}$$

The rule for Pads/Haskell parsers, and several of the other rules, use the function $valid(d)$. This function extracts a boolean from the metadata structure d, returning True if there are no errors in the structure and False otherwise.

The second and third semantic judgements specify the representation and metadata types for a given specification. They have the form $\mathcal{R}[\![s]\!] = \tau$ and $\mathcal{M}[\![s]\!] = \tau$, respectively. Note here that *att*

s	$\mathcal{R}[\![s]\!] =$	$\mathcal{M}[\![s]\!] =$
$k_{\tau_r}^{\tau_m}$	τ_r	$\tau_m\ md$
$\mathsf{Adhoc}(b_{\tau_r}^{\tau_m})$	τ_r	$(\tau_m \times att)\ md$
$e :: s$	$\mathcal{R}[\![s]\!]$	$\mathcal{M}[\![s]\!]$
$\langle x{:}s_1, s_2 \rangle$	$\mathcal{R}[\![s_1]\!] \times \mathcal{R}[\![s_2]\!]$	$(\mathcal{M}[\![s_1]\!] \times \mathcal{M}[\![s_2]\!])\ md$
$\{s \mid x \in e\}$	$\mathcal{R}[\![s]\!]\ list$	$(\mathcal{M}[\![s]\!]\ list)\ md$
$\mathsf{Pred}(e)$	$unit$	$unit\ md$
$s?$	$\mathcal{R}[\![s]\!]\ option$	$(\mathcal{M}[\![s]\!]\ option)\ md$

Figure 8. Forest calculus data and metadata types

is the type for file attribute records and the md type is defined as follows.

$$\begin{array}{lcl} \tau\ md & = & header \times \tau \\ header & = & bool \end{array}$$

The three sets of definitions obey the following basic coherence property, where $\vdash v : \tau$ states that the value v has type τ.

Proposition 1
If $\mathcal{E}; F; r \models s \rightsquigarrow v, d$ *and* $\mathcal{R}[\![s]\!] = \tau_{\mathcal{R}}$ *and* $\mathcal{M}[\![s]\!] = \tau_{\mathcal{M}}$ *then* $\vdash v : \tau_{\mathcal{R}}$ *and* $\vdash d : \tau_{\mathcal{M}}$

8. Related Work

The work in this paper builds upon ideas developed in the Pads project [3, 4]. 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 file systems. 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 [14] extends the .NET languages to enable querying any data source that supports the `IEnumerable` interface using a simple, convenient syntax. LINQ differs in that it does not provide support for declaratively specifying the structure of, and then ingesting, filestores. *Type Providers* [19], an experimental feature of F#, help programmers materialize standard data sources equipped with predefined schema (such as XML documents or databases) in memory in an F# program. Type Providers and Forest descriptions are complementary language features. In fact, it may be possible to define a new F# Type Provider capable of interpreting Forest file system schema and ingesting the described data, thereby making any Forest-described data available in F#.

In the databases community, a number of XML-based description languages have been defined for specifying file formats, file organization and file locations. One example of such a language is XFiles [1]. XFiles has many features in common with Forest. It can describe file locations, permissions, ownership and other attributes. It can also specify the name of an application capable of parsing the files in question. The main difference between a language like 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. XFiles does not inter-operate directly with conventional programming languages.

9. Conclusions

In this paper, we present the design of Forest, an embedded domain-specific language for describing filestores. A Forest description concisely specifies a collection of files, directories, and symbolic links as well as expected file system attributes such as owners and permissions. From a description, the Forest compiler generates code to lazily load the on-disk data into an isomorphic in-memory representation, lowering the divide between on-disk and in-memory data. Forest also generates type class instances that make it easy for third-party tool developers to use Haskell's generic programming infrastructure. We have used this infrastructure ourselves to define a number of useful tools. In addition, the language has a formal semantics based on classical tree logics and is fully implemented. On the latter point, our work serves as an extensive case study in domain-specific language design, and, as such, has inspired changes in the design of Template Haskell. We anticipate releasing the source code for Forest shortly.

References

[1] S.-C. Buraga. An XML-based semantic description of distributed file systems. In *RoEduNet International Conference on Networking in Education and Research*, pages 41–48, 2003.

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

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

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

[5] Forest Auxiliary Submission Materials, 2010. These materials will become available via the program chair after initial review is submitted.

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

[7] 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/`.

[8] 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.

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

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

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

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

[13] 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.

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

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

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

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

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

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

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