TxForest: A DSL for Concurrent Filestores

Jonathan DiLorenzo[1], Katie Mancini[1], Kathleen Fisher[2], and Nate Foster[1]

[1] Cornell University, Ithaca NY 14850, USA
[2] Tufts University, Medford MA 02155, USA

Abstract. Many systems use ad hoc collections of files and directories to store persistent data. For consumers of this data, the process of properly parsing, using, and updating these *filestores* using conventional APIs is cumbersome and error-prone. Making matters worse, most filestores are too big to fit in memory, so applications must process the data incrementally while managing concurrent accesses by multiple users. This paper presents Transactional Forest (TxForest), which builds on earlier work on Forest to provide a simpler, more powerful API for managing filestores, including a mechanism for managing concurrent accesses using serializable transactions. Under the hood, TxForest implements an optimistic concurrency control scheme using Huet's *zippers* to track the data associated with filestores. We formalize TxForest in a core calculus, develop a proof of serializability, and describe our OCaml prototype, which we have used to build several practical applications.

Keywords: Data description languages · File systems · Ad hoc data · Concurrency · Transactions · Zippers

1 Introduction

Modern database systems offer numerous benefits to programmers, including rich query languages and impressive performance. However, programmers in many areas including finance, telecommunications, and the sciences, rely on *ad hoc* data formats to store persistent data—e.g., flat files organized into structured directories. This approach avoids some of the initial costs of using a database such as writing schemas, creating user accounts, and importing data, but it also means that programmers must build custom tools for correctly processing the data—a cumbersome and error-prone task.

In many applications, multiple users must read and write the data stored on the file system concurrently, and even in settings where there is only a single user, parallelism can often be used to improve performance. For example, many instructors in large computer science courses rely on filestores to manage student data, encoding assignments, rosters, and grades as ad hoc collections of directories, CSVs, and ASCII files respectively. During grading, instructors use various scripts to manipulate the data—e.g., computing statistics, normalizing raw scores, and uploading grades to the registrar. However, these scripts are written against low-level file system APIs and rarely handle multiple concurrent users. This can easily lead to incorrect results or even data corruption in courses that use large numbers of TAs to help with grading.

The PADS/Forest family of languages offers a promising approach for managing ad hoc data. In these languages, the programmer specifies the structure of an ad hoc data format using a simple, declarative specification, and the compiler generates an in-memory representation for the data, load and store functions for mapping between in-memory and on-disk representations, as well as tools for analyzing, transforming, and visualizing the data. PADS focused on ad hoc data stored in individual files [6], while Forest handles ad hoc data in *filestores*—i.e., structured collections of files, directories, and links [5]. Unfortunately, the languages that have been proposed to date lack support for concurrency.

To address this challenge, this paper proposes Transactional Forest (TxForest), a declarative domain-specific language for correctly processing ad hoc data in the presence of concurrency. Like its predecessors, TxForest uses a type-based abstraction to specify the structure of the data and its invariants. From a TxForest description, the compiler generates a typed representation of the data as well as a high-level programming interface that abstracts away direct interactions with the file system and provides operations for automatically loading and storing data, while gracefully handling errors. TxForest also offers serializable transactions to help implement concurrent applications.

The central abstraction that facilitates TxForest's serializable semantics, as well as several other desired properties, is based on Huet's *zippers* [10]. Rather than representing a filestore in terms of the root node and its children, a zipper encodes the current node, the path traversed to get there, and the nodes encountered along the way. Importantly, local changes to the current node as well as common navigation operations involving adjacent nodes can be implemented in constant time. Additionally, by replacing the current node with a new value and then 'zipping' the tree back up to the root, modifications can be implemented in a purely functional way.

As others have also observed [11], zippers are a natural abstraction for filestores, for several reasons: (1) the concept of the working path is cleanly captured by the current node; (2) most operations are applied close to the current working path; (3) the zipper naturally captures incrementality by loading data as it is encountered in the zipper traversal; and (4) a traversal (along with annotations about possible modification) provides all of the information necessary to provide rich semantics, such as copy-on-write, as well as a simple optimistic concurrency control scheme that guarantees serializability.

In this paper, we first formalize the syntax and semantics of TxForest assuming a single thread of execution, and we establish various correctness properties, including roundtripping laws in the style of lenses [8]. Next, we extend the semantics to handle multiple concurrent threads, and introduce a transaction manager that implements a standard optimistic concurrency scheme. We prove that all transactions that sucessfully commit are serializable with respect to one another. Finally, we present a prototype implementation of TxForest as an

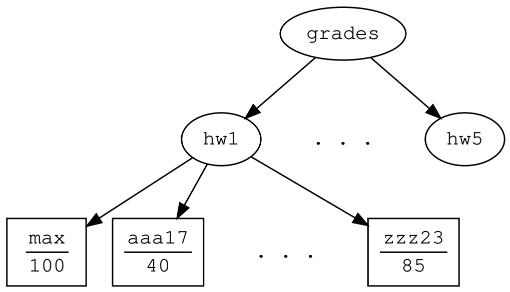

Fig. 1. Example: file system fragment used to store course data.

embedded language in OCaml, illustrating the feasibility of the design, and use it to implement several realistic applications.

Overall, the contributions of this paper are as follows:

– We present Transactional Forest, a declarative domain-specific language for processing ad hoc data in concurrent settings (Sections 3 and 4).
– We describe a prototype implementation of Transactional Forest as an embedded domain-specific language in OCaml (Section 5).
– We prove that our design satisfies several formal properties including round-tripping laws and serializability.

The rest of this paper is structured as follows: Section 2 introduces a simple example to motivate TxForest. Section 3 presents the syntax and single-threaded semantics of TxForest. Section 4 adds the multi-threaded semantics and the serializability theorem. Section 5 discusses the OCaml implementation of TxForest and an application. We review related work in Section 6 and conclude in Section 7. The proofs of formal properties are in the technical report [2].

2 Example: Course Management System

This section introduces an example of an idealized course management system to motivate the design of TxForest. Figure 1 shows a fragment of a filestore used in tracking student grades. The top-level directory (`grades`) contains a set of sub-directories, one for each homework assignment (`hw1`–`hw5`). Each assignment directory has a file for each student containing their grade on the assignment (e.g., `aaa17`), as well as a special file (`max`) containing the maximum score for that homework. Although this structure is simple, it closely resembles filestores that have actually been used to keep track of grades at several universities.

There are various operations that one might want to perform on this filestore, but to illustrate the challenges related to concurrency, we will focus on normalization. Normalization might be used to ensure that the grades for a particular homework fall between some specified limits or match a given distribution. We assume an idempotent operation `f` that takes assignment statistics and the current score as arguments and computes a normalized score.

OCaml Implementation. To start, let us see how we might write a renormalization procedure for this filestore in a general-purpose language—e.g., OCaml. For simplicity, the code relies on helper functions, which are explained below.

```
let renormalize f hw gmin =
  let hwDir = sprintf "grades/hw%d" hw in
  let gmax = get_score (hwDir ^/ "max") in
  let studentFiles = get_students hwDir in
  let (cmin, cmax) = get_min_and_max studentFiles in
  map_scores (f cmin cmax gmin gmax) studentFiles
```

The `renormalize` function takes as input a function to normalize individual scores (`f`), the identifier of a homework assignment (`hw`), and the minimum score to use when scaling scores (`gmin`). It retrieves the value from the `max` file, using the `get_score` helper, which reads the file and parses it into a score. Next, it retrieves the paths to every student file (`studentFiles`) and computes the minimum (`cmin`) and maximum (`cmax`) score over all students using a helper function (`get_min_and_max`), which again accesses data in the underlying file system. Finally, it maps the function `f` over each score (using the aggregate statistics), and writes the new score back to the file, again using a helper function to perform the necessary iteration (`map_scores`) and file writes.

Although this procedure is simple, there are several potential pitfalls that could arise because of its use of low-level file system APIs. For example, one of the files or directories might not exist or there might be extra files in the file system. The structure of the filestore might be malformed, or might change over time. Any of these mistakes could lead to run-time errors, or worse they might silently succeed but produce incorrect results. This implementation also suffers from a more insiduous set of problems related to concurrency. Consider what happens if multiple members of the course staff execute the renormalization procedure concurrently. If the stage that computes the minimum and maximum scores is interleaved with the stage that invokes `f` and writes the normalized values back to the file system, we could easily be left with a mangled filestore and incorrect results—something that would likely be difficult to detect, diagnose, and fix.

Classic Forest Implementation. Next, let us consider an implementation in Forest [5]. We start by specifying the structure of the filestore as follows:

```
  grades = [h :: hws | h <- matches RE "hw[0-9]+"]
  students = file
  hws = directory {
    max is "max" :: file;
    students is [s :: students | s <- matches RE "[a-z]+[0-9]+"];
  }
```

The `grades` specification describes the structure of the top-level directory: a list of homework directories, each containing a file named *max* and a list of `students` (each represented as a `file`[1]).

[1] By integrating with PADS [6], we could go a step further and specify the contents of the file as well—i.e. a single line containing an integer.

Given this specification, the Forest compiler generates an in-memory representation for the data, as well as associated functions for loading data from and storing data to the file system:

```
type students_rep = string
type hws_rep = { max : string; students : students_rep list}
type grades_rep = hws_rep list
type grades_md = hws_md list md
val grades_load : filepath -> grades_rep * grades_md
val grades_store : filepath -> grades_rep * grades_md -> unit
```

The `md` types store metadata including permissions and information about whether errors were encountered during loading. The `load` and `store` functions map between the on-disk and in-memory representations, and automatically check for errors and inconsistencies in the data. Using these functions, we can write the `renormalize` procedure as follows:

```
let renormalize f hw gmin : unit =
  let (gr,gmd) = grades_load (baseDir ^/ "grades") in
  if gmd.num_errors = 0 then
    let (hwr,hwmd) = find (sprintf "hw%d" hw) (gr,gmd) in
    let gmax = get_score hwr.max in
    let (cmin, cmax) = get_min_max hwr in
    map_scores (f cmin cmax gmin gmax) hwr hwmd
  else
    failwith (String.concat "\n" gmd.error_msg)
```

This code is similar to the OCaml implementation, but there are a few key differences. It first loads the entire `grades` directory and checks that it has no errors. This makes the auxilliary functions, like `get_score` (which now just turns a string into an integer) and `set_score` simpler and more robust, since they no longer need to worry about such issues. It then locates the representation and metadata for the assignment, computes aggregate statistics, and invokes `f` to renormalize and update the scores. The `get_min_max` and `map_scores` helpers are similar to the direct versions discussed previously.

The Forest implementation offers several important benefits over the OCaml code: (1) the structure of the filestore is explicit in the specification and the code; (2) the use of types makes certain programming mistakes impossible, such as attempting to read a file at a missing path; and (3) any part of the filestore not conforming to the specification is automatically detected.

However, the Forest code still suffers from the same concurrency issues discussed above. Further, it is unnecessary (and often infeasible) to load the entire filestore into memory—e.g., suppose we only need to manipulate data for a single homework or an individual student.

Transactional Forest Implementation. TxForest offers the same advantages as Forest, while dealing with issues related to concurrency and incrementality. The only cost is a small shift in programming style—i.e., navigating using a zipper.

The TxForest specification for our running example is identical to the Forest version. However, this surface-level specification is then translated to a core language (Section 3) that uses Huet's zipper internally and also provides transactional guarantees. The TxForest code for the `renormalize` function is different than the Forest version. Here is one possible implementation:

```
let renormalize f hw gmin zipper : (unit,string) Result.t =
  let%bind hwZ = goto_name_p (sprintf "hw%d" hw) zipper in
  let%bind gmax = goto_name_p "max" hwZ >>= get_score in
  let%bind studentZ = goto "students" hwZ in
  let%bind (cmin, cmax) = get_min_and_max studentZ in
  map_scores (f cmin cmax gmin gmax) studentZ
```

Note that the type of the function has changed so that it takes a zipper as an argument and returns a value in the result monad:

```
type ('a,'b) Result.t = Ok of 'a | Error of 'b
```

Intuitively, this monad tracks the same sorts of errors seen in the Forest code—e.g. from malformed filestores, but not from concurrency issues.

The `goto_name_p` function traverses the zipper—e.g., `goto_name_p "hw1" zipper` navigates to the comprehension node named *hw1* and then down to the corresponding file system path, ending up at a `hws` node. The *bind* operator (`>>=`) threads the resulting zipper through the monad. The `let%bind` x `=` e_1 `in` e_2 syntax is shorthand for e_1 `>>= fun` x `->` e_2. The `goto` function is similar to `goto_name_p`, but is limited to directories and does not walk down the last path operator. Finally, the helper functions, `map_scores` and `get_min_max`, use TxForest library functions to map and fold over the zipper respectively.

To use the `renormalize` function, users need some way to construct a zipper. The TxForest library provides functions called `run_txn` and `loop_txn`:

```
type txError = TxError | OpError of string
val run_txn : spec -> path -> (zipper -> ('a,string) Result.t) ->
              (unit -> ('a,txError) Result.t)
val loop_txn : spec -> path -> (zipper -> ('a,string) Result.t) ->
               (unit -> ('a,string) Result.t)
```

which might be used as follows:

```
match run_txn grades_spec "grades" (renormalize 1 60) () with
| Error TxError -> printf "Transaction aborted due to conflict"
| Error(OpError err) -> printf "Transaction aborted with error: %s" err
| Ok _ -> printf "Renormalization successful"
```

The `run_txn` function takes a specification, an initial path, and a function from zippers to results and produces a thunk. When the thunk is forced, it constructs a zipper focused on the given path and runs the function. If this execution results in an error, the outer computation produces an `OpError`. Otherwise, it attempts to commit the modifications produced during the computation. If the commit succeeds, it returns the result of the function, otherwise it discards the results and returns a `TxError`. The `loop_txn` function is similar, but retries the transaction until there is no conflict or the input function produces an error.

TxForest guarantees that transactions will be serializable with respect to other transactions—i.e., the final file system will be equivalent to one produced by executing the committed transactions in some serial order. See Section 4 for the formal concurrent semantics and the serializability theorem. In our example, this means that no errors can occur due to running multiple renormalization transactions simultaneously. Furthermore, TxForest automatically provides incrementality by only loading the data needed to traverse the zipper—an important property in larger filestores. Incremental Forest [3] provides a similar facility, but requires explicit user annotations. Overall, TxForest provides incremental support for filestore applications in the presence of concurrency. The next two sections present the language in detail, develop an operational model, and establish its main formal properties.

3 Transactional Forest

This section presents TxForest in terms of a core calculus. We discuss the goals and high level design decisions for the language before formalizing the syntax and semantics as well as several properties including round-tripping laws, equational identities, and consistency relations. Finally, we give a taste of the core calculus by using it to encode functions that would be useful for the course management example above. This section deals primarily with the single-threaded semantics, while the next section presents a concurrent model.

The main goals of this language are to allow practical processing of filestores for non-expert users. This leads to several requirements: (1) an intuitive way of specifying filestores [5]; (2) automatic, incremental processing, as filestores are typically large; (3) automatic concurrency control, as concurrency is both common and difficult to get right; and (4) transparency, as filestore interaction can be expensive and should therefore be explicit.

The zipper abstraction that our language is based on helps us achieve our second and fourth requirement. Both of these requirements and concurrency are then further addressed by our locality-centered language design: The semantics of every command and expression only considers the locale around the focus node of the zipper. This means that every command can restrict its attention to a small part of the filestore, which, along with the fact that data can be loaded as-required while traversing the zipper, gives us incrementality. We believe that the combination of locality and explicit zipper traversal commands also gives us transparency. In particular, the footprint of any command is largely predictable based on the filestore specification and current state. Predictability also simplifies tasks such as logging reads and writes, which is useful for concurrency control.

3.1 Syntax

In our formal model, we view a file system as a map from paths to file system contents, which are either directories (a set of their children's names) or files

Strings	u	$\in$	Σ^*
Integers	i	$\in$	$\mathbb{Z}$
Variables	x	$\in$	Var
Values	v	$\in$	Val
Environments	$E \in Env$	$:$	$Var \mapsto Val$
Paths	p	$::=$	$\texttt{/} \mid p/u$
Contents	T	$::=$	$\texttt{Dir}\ \{\overline{u}\} \mid \texttt{File}\ u$
File Systems	fs	$:$	$Path \mapsto Content$
Contexts	$ctxt$	$:$	$Env \times Path \times 2^{Path} \times Zipper$

Fig. 2. Preliminaries

(strings). For a path and file system, p and fs, we define $p \in fs \triangleq p \in \texttt{dom}(fs)$. See Figure 2 for the metavariable conventions used in our formalization. We assume that all file systems are well formed—i.e., that they encode a tree, where each node is either a directory or a file with no children:

Definition 1 (Well-Formedness). *A file system fs is well-formed iff:*

1. $fs(\texttt{/}) = \texttt{Dir}\ \_$ *(where* / *is the root node)*
2. $p/u \in fs \iff fs(p) = \texttt{Dir}\ \{u; \ldots\}$

In this definition, the notation _ indicates an irrelevant hole which may be filled by any well-typed term. We use this convention throughout the paper.

In the previous section, we gave a flavor of the specifications one might write in TxForest. We wrote these specifications in our surface language, which compiles down to a core calculus, whose syntax is given in Figure 3. The core specifications are described fully below, but first, we will provide the translation of the `hws` specification from Section 2 to provide an intuition:

```
directory {
    max is "max" :: file;
    students is [s :: students | s <- matches RE "[a-z]+[0-9]+"]
}
```

becomes

$\langle max : "max" :: File, \langle dir : Dir, [s :: students \mid s \in e]\rangle\rangle$

```
where e = filter (Run Fetch_Dir dir) "[a-z]+[0-9]+"
```

Directories become dependent pairs, allowing earlier parts of directories to be referenced by later parts. Comprehensions, which use regular expressions to query the file system, also turn into dependent pairs: The first component of the pair is a *Dir*. The second component fetches from and filters the first component using a regular expression. Section 3.4 gives examples of functions written against this specification using the command language described below. We proceed by describing the syntax shown in Figure 3 in-depth.

Formally, a TxForest specification s describes the shape and contents of a *filestore*, which is a structured subtree of a file system. Such specifications are almost identical to those in Classic Forest [5]. To a first approximation, they can be understood as follows:

Specifications	$s \in spec$	$::=$	$File \mid Dir \mid e :: s \mid \langle x : s_1, s_2 \rangle$
		$\mid$	$[s \mid x \in e] \mid s? \mid P(e)$
Zippers	z	$::=$	$\{$`ancestor` : $Zipper$ `option`;
			`left` : $(Env \times spec)$ `list`;
			`current` : $(Env \times spec)$;
			`right` : $(Env \times spec)$ `list`$\}$
Commands	c	$::=$	$fc \mid$ `Skip` $\mid c_1; c_2 \mid x := e$
		$\mid$	`If` b `Then` c_1 `Else` $c_2 \mid$ `While` b `Do` c
Forest Commands	fc	$::=$	$fn \mid fu$
Forest Navigations	fn	$::=$	`Down` $\mid$ `Up` $\mid$ `Next` $\mid$ `Prev`
		$\mid$	`Into_Pair` $\mid$ `Into_Comp` $\mid$ `Into_Opt` $\mid$ `Out`
Forest Updates	fu	$::=$	`Store_File` $e \mid$ `Store_Dir` $e \mid$ `Create_Path`
Expressions	e, b	$::=$	$fe \mid v \mid x \mid e_1\ e_2 \mid \ldots$
Forest Expressions	fe	$::=$	`Fetch_File` $\mid$ `Fetch_Dir` $\mid$ `Fetch_Path`
		$\mid$	`Fetch_Comp` $\mid$ `Fetch_Opt` $\mid$ `Fetch_Pred`
		$\mid$	`Run` $fn\ e \mid$ `Run` $fe\ e \mid$ `Verify`
Log Entries	le	$::=$	`Write_file` $T_1\ T_2\ p \mid$ `Read` $T\ p$
		$\mid$	`Write_dir` $T_1\ T_2\ p$
Logs	σ	$:$	$LogEntry$ `list`
Programs	g	$::=$	(p, s, c)

Fig. 3. Main Syntax

– *Files and Directories.* The *File* and *Dir* specifications describe filestores with a file and directory, respectively, at the current path.
– *Paths.* The $e :: s$ specification describes a filestore modeled by s at the extension of the current path denoted by e.
– *Dependent Pairs.* The $\langle x : s_1, s_2 \rangle$ specification describes a filestore modeled by both s_1 and s_2. Additionally, s_2 may use the variable x to refer to the portion of the filestore matched by s_1.
– *Comprehensions.* The $[s \mid x \in e]$ specification describes a filestore modeled by s when x is bound to any element in the set e.
– *Options.* The $s?$ specification describes a filestore that is either modeled by s or where the current path does not exist.
– *Predicates.* The $P(e)$ specification describes a filestore where the boolean e is `true`. This construct is typically used with dependent pairs.

Most specifications can be thought of as trees with as many children as they have sub-specifications. Comprehensions are the exception; we think of them as having as many children as there are elements in the set e.

To enable incremental and transactional manipulation of data contained in filestores, TxForest uses a *zipper* which is constructed from a specification. The zipper traverses the specification tree while keeping track of an environment that binds variables from dependent pairs and comprehensions. The zipper can be thought of as representing a tree along with the particular node of the tree that is in focus. We use the symbol `current` to represents the focus node, while `left` and `right` represent its siblings to the left and right respectively. The symbol `ancestor` tracks the focus node's ancestors by containing the zipper we came from before moving down to this depth of the tree. Key principles to keep in mind regarding zippers are that (1) the tree can be unfolded as it is traversed and (2) operations near the current node are fast, thus optimizing for locality.

To express navigation on the zipper, we use standard imperative (IMP) commands, c, as well as special-purpose Forest Commands, fc, which are divided into Forest Navigations, fn, and Forest Updates, fu. Navigation commands are those that traverse the zipper, while Update commands modify the file system. Expressions are mostly standard and pure: they never modify the file system and only Forest Expressions query it. Forest Commands and Expressions will be described in greater detail in Section 3.2. To ensure serializability among multiple TxForest threads executing concurrently, we will maintain a log. An entry `Read` T p indicates that we have read T at path p while `Write_file` T_1 T_2 p (respectively `Write_dir` T_1 T_2 p) indicates that we have written the file (respectively directory) T_2 to path p, where T_1 was before.

3.2 Semantics

Having defined the syntax, we now present the denotational semantics of TxForest. The semantics of IMP commands are standard and thus elided. We start by defining the semantics of a program:

$$[\![(p, s, c)]\!]_g\ fs \ \triangleq\ \texttt{project\_fs}\ ([\![c]\!]_c\ (\{\}, p, \{\}, \wr\{\}, s\int)\ fs)$$

The denotation of a TxForest program is a function on file systems. We use the specification s to construct a new zipper, seen in the figure using our zipper notation defined after this paragraph. Then we execute the command c using the denotation of commands $[\![\cdot]\!]_c$. The denotation function takes a context, which we construct using the zipper and the path p, and a file system fs as arguments. The denotation function then produces a new context and file system, from which we project out the file system with `project_fs`.

Definition 2 (Zipper Notation). *We define notation for constructing and deconstructing zippers. To construct a zipper we write*
$\mathit{left} \leftharpoonup \wr\mathit{current}\int^{z} \rightharpoonup \mathit{right} \triangleq \{\mathit{ancestor} = \mathit{Some}(z); \mathit{left}; \mathit{current}; \mathit{right}\}$, *where any of* $\mathit{ancestor}$, left, *and* right *can be omitted to denote a zipper with* $\mathit{ancestor} = \mathit{None}$, $\mathit{left} = []$, *and* $\mathit{right} = []$ *respectively. For example:*

$$\wr\mathit{current}\int \triangleq \{\mathit{ancestor} = \mathit{None}; \mathit{left} = []; \mathit{current}; \mathit{right} = []\}$$

Likewise, to destruct a zipper we write $\mathit{left} \leftharpoonup (\!|\mathit{current}|\!)^{z} \rightharpoonup \mathit{right}$ *where any part can be omitted to ignore that portion of the zipper, but any included part must exist. For example,* $z = (\!|\_|\!)^{z'} :\Longleftrightarrow z.\mathit{ancestor} = \mathit{Some}(z')$.

Command: fc	Conditions: Φ	Def. of $[\![fc]\!]_c\ (E, p, ps, z)\ fs$ when Φ
`Down`	$z = \llparenthesis E_L, e :: s \rrparenthesis$ $(u, \sigma) = [\![e]\!]_e\ (E_L, p, ps, z)\ fs$ $\texttt{Dir}\ \ell = fs(p)$	$\sigma' = \sigma \cdot (\texttt{Read}\ (\texttt{Dir}\ \ell)\ p)$ $((E, p/u, ps \cup (p/u), \lbag E_L, s \rbag^{z}), fs, \sigma')$
`Up`	$z = \llparenthesis \_ \rrparenthesis^{z'}$ $z' = \llparenthesis \_, e :: s \rrparenthesis$	$((E, \texttt{pop}\ p, ps, z'), fs, \epsilon)$
`Into_Opt`	$z = \llparenthesis E_L, s? \rrparenthesis$	$((E, p, ps, \lbag E_L, s \rbag^{z}), fs, \epsilon)$
`Into_Pair`	$z = \llparenthesis E_L, \langle x : s_1, s_2 \rangle \rrparenthesis$	$ctxt = (E_L, p, ps, \lbag E_L, s_1 \rbag)$ $z' = \lbag E_L, s_1 \rbag^{z} \rightharpoondown [(E_L[x \mapsto ctxt], s_2)]$ $((E, p, ps, z'), fs, \epsilon)$
`Into_Comp`	$z = \llparenthesis E_L, [s \mid x \in e] \rrparenthesis$ $(h \cdot t, \sigma) = [\![e]\!]_e\ (E_L, p, ps, z)\ fs$	$r\ \ = \texttt{map}\ (\lambda u.\ (E_L[x \mapsto u], s))\ t$ $z' = \lbag E_L[x \mapsto h], s \rbag^{z} \rightharpoondown r$ $((E, p, ps, z'), fs, \sigma)$
`Out`	$z = \llparenthesis \_ \rrparenthesis^{z'}$ $z' \neq \llparenthesis \_, e :: s \rrparenthesis$	$((E, p, ps, z'), fs, \epsilon)$
`Next`	$z = l \leftharpoonup \llparenthesis c' \rrparenthesis^{z'} \rightharpoondown (c \cdot r)$	$z'' = (c' \cdot l) \leftharpoonup \lbag c \rbag^{z'} \rightharpoondown r$ $((E, p, ps, z''), fs, \epsilon)$
`Prev`	$z = (c \cdot l) \leftharpoonup \llparenthesis c' \rrparenthesis^{z'} \rightharpoondown r$	$z'' = l \leftharpoonup \lbag c \rbag^{z'} \rightharpoondown (c' \cdot r)$ $((E, p, ps, z''), fs, \epsilon)$
`Store_File` e	$z = \llparenthesis \_, File \rrparenthesis$ $(u, \sigma) = [\![e]\!]_e\ (E, p, ps, z)\ fs$	$(fs', \sigma') = \texttt{make\_file}\ fs\ p\ u$ $((E, p, ps, z), fs', \sigma \cdot \sigma')$
`Store_Dir` e	$z = \llparenthesis \_, Dir \rrparenthesis$ $(\ell, \sigma) = [\![e]\!]_e\ (E, p, ps, z)\ fs$	$(fs', \sigma') = \texttt{make\_directory}\ fs\ p\ \ell$ $((E, p, ps, z), fs', \sigma \cdot \sigma')$
`Create_Path`	$z = \llparenthesis E_L, e :: s \rrparenthesis$ $(u, \sigma) = [\![e]\!]_e\ (E_L, p, ps, z)\ fs$	$(fs', \sigma') = \texttt{create}\ fs\ p/u$ $\sigma'' \quad = \sigma \cdot (\texttt{Read}\ fs(p)\ p) \cdot \sigma'$ $((E, p, ps, z), fs', \sigma'')$

Fig. 4. *fc* Command Semantics

The two key invariants that hold during the execution of any command are (1) that the file system remains well-formed (Definition 1) and (2) that if $[\![fc]\!]_c\ (\_, p/u, \_, \_)\ fs = ((\_, p'/u', \_, \_), fs', \_)$ and $p \in fs$, then $p' \in fs'$. The first property states that no command can make a well-formed file system ill-formed. The second states that, as we traverse the zipper, we maintain a connection to the real file system. It is important that only the parent of the current file system node is required to exist as this allows us to construct new portions of the filestore and handle the option specification. A central design choice that underpins the semantics is that each command acts *locally* on the current zipper and does not require further context. This makes the cost of the operation apparent and, as in Incremental Forest [3], facilitates partial loading and storing. These properties can be seen from Figure 4 which defines the semantics of Forest Commands.

As illustrated in the top row of the table, each row should be interpreted as defining the meaning of evaluating a command in a given context, (E, p, ps, z), and file system, *fs*, provided the conditions hold. The denotation function is partial, being undefined if none of the rows apply. Intuitively, a command is undefined when it is used on a malformed filestore with respect to its specification, or when it is ill-typed—i.e. used on an unexpected zipper state. Operationally, the semantics of each command can be understood as follows:

– `Down` and `Up` are duals: the first traverses the zipper into a path expression, simultaneously moving us down in the file system, while the other does the reverse. Additionally, `Down` queries the file system, producing a `Read`.
– `Into` and `Out` are duals: the first traverses the zipper into its respective type of specification, while the second moves back out to the parent node. Additionally, their subexpressions may produce logs.
  For dependent pairs, we update the environment of the second child with a context constructed from the first specification.
  For comprehensions, the traversal requires the set denoted by e to be non-empty, and maps it to a list of children with the same specification, but environments with different mappings for x, before moving to the first child.
– `Next` and `Prev` are duals: the first traverses the zipper to the right sibling and the second to the left sibling.
– `Store_File` e, `Store_Dir` e, and `Create_Path` all update the file system, leaving the zipper untouched. All of the functions they call out to close the file system to remain well-formed and their definitions can be found in the technical report [2]. These functions produce logs recording their effects.
  For `Store_File` e, e must evaluate to a string, u, after which the command turns the current file system node into a file containing u.
  For `Store_Dir` e, e must evaluate to a string set, ℓ, after which the command turns the current file system node into a directory containing that set. If the node is already a directory containing ℓ', then any children in $\ell' \setminus \ell$ are removed, any children in $\ell \setminus \ell'$ are added (as empty files) and any children in $\ell \cap \ell'$ are untouched.
  For `Create_Path`, the current node is turned into a directory containing the path that the path expression points to. The operation is idempotent and does the minimal work required: If the current node is already a directory, then the path is added. If the path was already there, then `Create_Path` is a no-op, otherwise it will map to an empty file.

With that, we have covered the semantics of all of the Forest Commands, but their subexpressions remain. The semantics of non-standard expressions is given in Figure 5. The interpretation of each row is the same as for commands. There is one `Fetch` expression per specification except for pairs, which have no useful information available locally. Since a pair is defined in terms of its sub-specifications, we must navigate to them before fetching information from them. This design avoids incurring the cost of eagerly loading a large filestore.

Fetching a file returns the string contained by the file at the current path. For a directory, we get the names of its children. Both of these log `Read`s since they inspect the file system. For a path specification, the only locally available

Expression: fe	Conditions: Φ	Def. of $[\![fe]\!]_e\ (E,p,ps,z)\ fs$ when Φ
`Fetch_File`	$z = (\!\|\_, File\|\!)$ $\texttt{File}\ u = fs(p)$	$(u, [\texttt{Read}\ (\texttt{File}\ u)\ p])$
`Fetch_Dir`	$z = (\!\|\_, Dir\|\!)$ $\texttt{Dir}\ \ell = fs(p)$	$(\ell, [\texttt{Read}\ (\texttt{Dir}\ \ell)\ p])$
`Fetch_Path`	$z = (\!\|E_L, e :: s\|\!)$	$[\![e]\!]_e\ (E_L,p,ps,z)\ fs$
`Fetch_Comp`	$z = (\!\|E_L, [s \mid x \in e]\|\!)$	$[\![e]\!]_e\ (E_L,p,ps,z)\ fs$
`Fetch_Opt`	$z = (\!\|\_, s?\|\!)$	$(p \in fs, [\texttt{Read}\ fs(p)\ p])$
`Fetch_Pred`	$z = (\!\|E_L, P(e)\|\!)$	$[\![e]\!]_e\ (E_L,p,ps,z)\ fs$
`Run` $fn\ e$	$(ctxt', \sigma') = [\![e]\!]_e\ (E,p,ps,z)\ fs$ $(ctxt, fs, \sigma) = [\![fn]\!]_c\ ctxt'\ fs$	$(ctxt, \sigma' \cdot \sigma)$
`Run` $fe\ e$	$(ctxt, \sigma') = [\![e]\!]_e\ (E,p,ps,z)\ fs$ $(v, \sigma) = [\![fe]\!]_e\ ctxt\ fs$	$(v, \sigma' \cdot \sigma)$
`Verify`	`true`	$(p', z') = \texttt{goto\_root}\ (E,p,ps,z)\ fs$ $\texttt{PConsistent}\ (p', ps, z')\ fs$

Fig. 5. Expression Semantics

information is the actual path. For a comprehension, we return the set e. For an option, we determine whether the current path is in the file system and log a `Read` regardless. Finally, for a predicate, we determine if its condition holds.

There are two `Run` expressions. The subexpression, e, must evaluate to a context. These can only come from a dependent pair, which means that `Run`s can only occur as subexpressions of specifications. We utilize them by performing traversals (`Run` $fn\ e$) and evaluating Forest expressions (`Run` $fe\ e$) in the input context. For example, a filestore defined by a file `index.txt` and a set of files listed in that index could be described as follows:

$\langle index : \text{"}index.txt\text{"} :: File, [x :: File \mid x \in e]\rangle$
`where` e `= lines_of (Run Fetch_File (Run Down` $index$`))`

where `lines_of` maps a string to a string set by splitting it by lines.

Finally, `Verify` checks the partial consistency of the traversed part of the filestore—i.e. whether it conforms to our specification. Unfortunately, checking the entire filestore, even incrementally can be very expensive and, often, we have only performed some local changes and thus do not need the full check. Partial consistency is a compromise wherein we only check the portions of the filestore that we have traversed, as denoted by the path set. This ensures that the cost of the check is proportional to the cost of the operations we have already run. Partial consistency is formally defined in the next subsection, which among other properties, details the connection between partial and full consistency.

3.3 Properties

This section establishes properties of the TxForest core calculus: consistency and partial consistency, equational identities on commands, and round-tripping laws.

Spec: s	Conditions: Φ	Def. of $\texttt{PConsistent}\ (p, ps, \llparenthesis(E,s)\rrparenthesis \text{ as } z)\ fs$ when Φ
$\_$	$p \notin ps$	$((\texttt{true}, \texttt{false}), \epsilon)$
File	$p \in ps$	$((fs(p) = \texttt{File}\ \_, \texttt{true}), [\texttt{Read}\ fs(p)\ p])$
Dir	$p \in ps$	$((fs(p) = \texttt{Dir}\ \_, \texttt{true}), [\texttt{Read}\ fs(p)\ p])$
$e :: s$	$p \in ps$	$(u, \sigma) = [\![e]\!]_e\ (E, p, ps, z)\ fs$ $((fs(p) = \texttt{Dir}\ \_, \texttt{true}), \sigma \cdot (\texttt{Read}\ fs(p)\ p))\ \wedge_\sigma$ $\texttt{PConsistent}\ (p/u, ps, \wr(E,s)\wr^{z})\ fs$
$\langle x : s_1, s_2 \rangle$	$p \in ps$	$ctxt = (E, p, ps, \wr(E, s_1)\wr)$ $E' \ = E[x \mapsto ctxt]$ $\texttt{PConsistent}\ (p, ps, \wr(E, s_1)\wr^{z} \rightharpoonup [(E', s_2)])\ fs\ \wedge_\sigma$ $\texttt{PConsistent}\ (p, ps, [(E, s_1)] \leftharpoonup \wr(E', s_2)\wr^{z})\ fs$
$[s \mid x \in e]$	$p \in ps$	$(\ell, \sigma') \qquad = [\![e]\!]_e\ (E, p, ps, z)\ fs$ $((b_1, b_2), \sigma) = \bigwedge_{v \in \ell} \texttt{PConsistent}\ (p, ps, \wr(E[x \mapsto v], s)\wr^{z})\ fs$ $((b_1, b_2), \sigma' \cdot \sigma)$
$s?$	$p \in ps$	$((p \notin fs, \texttt{true}), [\texttt{Read}\ fs(p)\ p])\ \vee_\sigma$ $\texttt{PConsistent}\ (p, ps, \wr(E,s)\wr^{z})\ fs$
$P(e)$	$p \in ps$	$(b, \sigma) = [\![e]\!]_e\ (E, p, ps, z)\ fs$ $((b, \texttt{true}), \sigma)$

$$
\begin{array}{lcl}
((\texttt{false}, \_), \sigma)\ \wedge_\sigma\ \_ & \triangleq & ((\texttt{false}, \texttt{false}), \sigma) \\
((b_1, b_2), \sigma)\ \wedge_\sigma\ ((b_1', b_2'), \sigma') & \triangleq & ((b_1\ \wedge\ b_1', b_2\ \wedge\ b_2'), \sigma \cdot \sigma') \\
((\texttt{true}, \texttt{true}), \sigma)\ \vee_\sigma\ \_ & \triangleq & ((\texttt{true}, \texttt{true}), \sigma) \\
((b_1, b_2), \sigma)\ \vee_\sigma\ ((b_1', b_2'), \sigma') & \triangleq & ((b_1\ \vee\ b_1', b_2\ \vee\ b_2'), \sigma \cdot \sigma') \\
\\
\texttt{complete?}\ ((\_, b), \_) & \triangleq & b \\
\texttt{consistent?}\ ((b, \_), \_) & \triangleq & b \\
\\
\texttt{Cover}\ (p, ps, z)\ fs & :\Longleftrightarrow & \texttt{complete?}\ (\texttt{PConsistent}\ (p, ps, z)\ fs)
\end{array}
$$

Fig. 6. Partial Consistency and Cover

The formal definition of partial consistency is given in Figure 6. Intuitively, full consistency (`Consistent`) captures whether a filestore conforms to its specification. For example, the file system, fs, at p conforms to *File* if and only if $fs(p) = \texttt{File}\ \_$ and to $e :: s$ if e evaluates to u and fs at p/u conforms to s. Partial consistency (`PConsistent`) then checks partial conformance (i.e. does the filestore conform to part of its specification). `PConsistent` returns two booleans (and a log), the first describing whether the input filestore is consistent with the input specification and the second detailing whether that consistency is total or partial. The definition of full consistency is very similar to partial, except that there are no conditions and the path set is ignored. The properties below describe the relationship between partial consistency and full consistency. Their proofs can be found in the technical report [2].

Theorem 1. *Consistency implies partial consistency:*

$$\begin{aligned}&\forall ps.\ \textit{consistent?}\ (\textit{Consistent}\ (p, ps, z)\ fs) \implies\\ &\quad \textit{consistent?}\ (\textit{PConsistent}\ (p, ps, z)\ fs)\end{aligned}$$

Theorem 2. *Partial Consistency is monotonic w.r.t. the path set:*

$$\begin{aligned}&\forall ps_1, ps_2.\ ps_2 \subseteq ps_1 \implies\\ &\quad \textit{consistent?}\ (\textit{PConsistent}\ (p, ps_1, z)\ fs) \implies\\ &\quad \textit{consistent?}\ (\textit{PConsistent}\ (p, ps_2, z)\ fs)\\ &\ \wedge\ \textit{complete?}\ (\textit{PConsistent}\ (p, ps_2, z)\ fs) \implies\\ &\quad \textit{complete?}\ (\textit{PConsistent}\ (p, ps_1, z)\ fs)\end{aligned}$$

This theorem says that if ps_1 is partially consistent, then any path set, ps_2, that is a subset of ps_1 will also be partially consistent. Conversely, if the consistency of ps_2 is total, or complete, then ps_1 will also be totally consistent.

Theorem 3. *Given a specification s and a path set ps that covers the entirety of s, partial consistency is exactly full consistency:*

$$\begin{aligned}&\forall ps.\ \exists ps'.\ \textit{Cover}\ (p, ps', z)\ fs\ \wedge\ ps' \subseteq ps \implies\\ &\quad \textit{consistent?}\ (\textit{Consistent}\ (p, ps, z)\ fs) \iff\\ &\quad \textit{consistent?}\ (\textit{PConsistent}\ (p, ps, z)\ fs)\end{aligned}$$

This theorem says that if the path set, ps is a superset of one that covers the entirety of the filestore, ps', as defined in Figure 6, then the filestore is totally consistent exactly when it is partially consistent. Intuitively, if a path set covers a filestore then we can never encounter a path outside of the path set while traversing the zipper.

Other properties of the language include identities of the form $[\![\texttt{Down};\texttt{Up}]\!]_c \equiv [\![\texttt{Skip}]\!]_c$ where $\equiv$ denotes equivalence modulo log when defined. That is, either $[\![\texttt{Down};\texttt{Up}]\!]_c$ is undefined, or it has the same action as $[\![\texttt{Skip}]\!]_c$, barring logging. Additionally, we have proven round-tripping laws in the style of lenses [8] stating, for example, that storing just loaded data is equivalent to `Skip`. Further identities and formal statements of these laws can be found in the technical report [2].

3.4 Examples

This subsection details the core calculus encodings of a few useful functions for interfacing with the course management system introduced in Section 2. The goal is to build an intuition for the language and how programming against the zipper abstraction might look. In practice, one would compile a higher-level language down to this core calculus for ease of use.

For the purposes of these examples, we will assume that `in` variables contain our input arguments at the start of each function and that `out` should contain the output of the function, if any, at the end. Additionally, all of our examples will be written against the same single-homework specification that we saw earlier in both our higher-level description language and in the core calculus:

```
directory {
    max is "max" :: file;
    students is [s :: students | s <- matches RE "[a-z]+[0-9]+"]
}
```

that is,

$\langle max : "max" :: File, \langle dir : Dir, [s :: students \mid s \in e]\rangle\rangle$
where e = filter (Run Fetch_Dir dir) "[a-z]+[0-9]+"

With that said, we will proceed to encode simple primitive functions for getting and setting the score of a single student and adding a student, a fold function over path comprehensions and finally a function for getting the average score of all students for a single homework.

```
getScore := λ(). to_int Fetch_File
setScore ≜ Store_File (of_int in)
```

In getScore and setScore, we assume that the zipper is already at a student. getScore, which we can define as an expression in the language, takes a unit input and fetches the current file, converting the string to an integer. setScore, like the rest of our examples, is instead a metavariable representing a particular command. This command converts *in* to a string before storing it as a file.

```
addStudent ≜
  Into_Pair; Next; Into_Pair;      # Go to dir
  Store_Dir (Fetch_Dir ∪ {in}); # Add in to the directory
  Out; Prev; Out                   # Return
```

In addStudent, we start from the root of the filestore and navigate to the first component of the internal pair. We then fetch the names of the current files in the directory before adding *in* and storing it back. Finally, we return to the root.

```
fold ≜
  $num$ := length Fetch_Comp; Into_Comp;
  While $num > 0$ Do
    Down;                    # Enter path
    $inAcc$ := $inF$ $inAcc$;      # Execute function and update accumulator
    Up; Next; $num$ := $num - 1$ # Go to next element
  Out;
  $out$ := $inAcc$
```

In fold, the zipper should start at a comprehension whose subspecification is a path expression. We take two inputs: *inAcc*, which is the initial accumulator value, and *inF*, which is a function that produces a new accumulator from the old one. The code for fold starts by getting the number of elements in the comprehension, before traversing the elements one by one and calling *inF* to update the accumulator at each element.

Finally, getAvg computes the average score across all students:

$$\begin{array}{rl} & \vdots \\ ts \in \mathit{Timestamp} & \text{Timestamps} \\ GL \in \mathit{TSLog} & \text{Timestamped Logs} \\ td \in \mathit{Thread} & \triangleq \mathit{Context} \times \mathit{Filesystem} \times \mathit{Command} \\ TxS \in \mathit{TxState} & \triangleq \mathit{Command} \times \mathit{Timestamp} \times \mathit{Log} \\ t \in \mathit{Transaction} & \triangleq \mathit{Thread} \times \mathit{TxState} \\ T \in \mathit{Thread\ Pool} & \triangleq \mathit{Transaction\ Bag} \end{array}$$

Fig. 7. Global Semantics Additional Syntax

```
getAvg ≜
  Into_Pair; Next; Into_Pair; Next;
  number := length Fetch_Comp;
  inAcc := 0;
  inF := λx. getScore () + x;
  fold;
  Prev; Out; Prev; Out;
  out := out / number
```

It starts at the root of the filestore and navigates to the comprehension. Next, it stores the number of students in *number*, sets *inAcc* to 0 and constructs *inF*, which gets the score of the current student and adds it to its argument. Then it folds, returns to the root of the filestore and divides the result of the fold (*out*) by the number of students to obtain the final result.

4 Concurrency Control

This section introduces the global semantics of Transactional Forest, using both a denotational semantics to concisely capture a serial semantics, and an operational semantics to capture thread interleavings and concurrency. We also state a serializability theorem that relates the two semantics.

Figure 7 lists the additional syntax used in this section. Timestamped logs are the logs of the global semantics. They are identical to local logs except that each entry also contains a timestamp signifying when it was written to the log.

Each *Thread* is captured by its local context, which, along with its transactional state, *TxState*, denotes a *Transaction*. The transactional state has 3 parts: (1) the command the transaction is executing; (2) the time when the transaction started; and (3) the transaction-local log recorded so far.

Our global denotational semantics is defined as follows:

$$\begin{aligned} [\![((ctxt, \_, c), \_)]\!]_G\ fs &\triangleq \texttt{project\_fs}\ ([\![c]\!]_c\ ctxt\ fs) \\ [\![\ell]\!]_{\mathbb{G}}\ fs &\triangleq \texttt{fold}\ fs\ \ell\ [\![\cdot]\!]_G \end{aligned}$$

The denotation of one or more transactions is a function on file systems. For a single transaction, it is the denotation of the command with the encapsulated

$$\frac{\langle td\rangle \xrightarrow{\sigma'}_L \langle td'\rangle}{\langle FS, GL, \{(td,(cs,ts,\sigma))\} \uplus T\rangle \rightarrow_G \langle FS, GL, \{(td',(cs,ts,\sigma\cdot\sigma'))\} \uplus T\rangle}$$

$$\frac{\begin{array}{c}\texttt{is\_Done?}\ td \qquad \texttt{check\_log}\ GL\ \sigma\ ts \\ FS' = \texttt{merge}\ FS\ \sigma \qquad GL' = GL \cdot (\texttt{add\_ts}\ \texttt{fresh\_ts}\ \sigma)\end{array}}{\langle FS, GL, \{(td,(cs,ts,\sigma))\} \uplus T\rangle \rightarrow_G \langle FS', GL', T\rangle}$$

$$\frac{\begin{array}{c}\texttt{is\_Done?}\ td \qquad \neg(\texttt{check\_log}\ GL\ \sigma\ ts) \qquad ts' = \texttt{fresh\_ts} \\ (z',p') = \texttt{goto\_root}\ (E,p,ps,z)\ fs \qquad td' = ((\{\},p',\{\},z'),FS,cs)\end{array}}{\langle FS, GL, \{(td,(cs,ts,\sigma))\} \uplus T\rangle \rightarrow_G \langle FS, GL, \{(td',(cs,ts',[]))\} \uplus T\rangle}$$

Fig. 8. Global Operational Semantics

context except for the file system which is replaced by the input. For a list of transactions, it is the result of applying the local denotation function in serial order. Note that the denotation of a transaction is precisely the denotation of a program, $[\![\cdot]\!]_g$, which can be lifted to multiple programs by folding. The key point to note about this semantics is that there is no interleaving of transactions. By definition, the transactions are run sequentially. While this ensures serializability, it also does not allow for any concurrency.

We will instead use an operational semantics that more easily models thread interleaving and prove that it is equivalent to the denotational semantics. First, we introduce an operational semantics for local commands. This semantics is standard for IMP commands, but for Forest Commands, it uses the denotational semantics, considering each a single atomic step, as seen below:

$$\frac{((E',p',ps',z'),fs',\sigma) = [\![fc]\!]_c\ (E,p,ps,z)\ fs}{\langle (E,p,ps,z),fs,fc\rangle \xrightarrow{\sigma}_L \langle (E',p',ps',z'),fs',\texttt{Skip}\rangle}$$

Next, we can construct the global operational semantics, as seen in Figure 8. The global stepping relation is between two global contexts which have three parts: A global file system, a global log, and a thread pool, or bag of transactions.

There are only three actions that the global semantics can take:

1. A transaction can step in the local semantics and append the resulting log.
2. A transaction that is done, and does not conflict with previously committed transactions, can commit. It must check that none of its operations conflicted with those committed since its start. Conflicts occur when the transaction read stale data. Then, it will update the global file system according to any writes performed. Finally, the transaction will leave the thread pool. The definitions of `check_log` and `merge` can be found in Figure 9.
3. A transaction that is done, but conflicts with previously committed transactions, cannot commit and instead has to restart. It does this by getting a fresh timestamp and resetting its log and local context.

In the operational semantics, thread steps can be interleaved arbitrarily, but changes will get rolled back in case of a conflict. Furthermore, while Forest

$$\begin{aligned}
\texttt{merge}\ FS\ \sigma &\triangleq \texttt{fold}\ FS\ \sigma\ \texttt{update}\\[4pt]
\texttt{update}\ fs\ (\texttt{Read}\ T\ p) &\triangleq fs\\
\texttt{update}\ fs\ (\texttt{Write\_file}\ \_\ T\ p) &\triangleq \texttt{close\_fs}\ (fs[p \mapsto T])\\
\texttt{update}\ fs\ (\texttt{Write\_dir}\ \_\ T\ p) &\triangleq \texttt{close\_fs}\ (fs[p \mapsto T])\\[4pt]
\texttt{check\_log}\ GL\ \sigma\ ts &\triangleq \forall p' \in \texttt{extract\_paths}\ \sigma.\ \forall (ts', le) \in GL.\\
&\quad\ ts > ts' \ \vee\ \neg(\texttt{conflict\_path}\ p'\ le)\\
\texttt{conflict\_path}\ p'\ (\texttt{Read}\ \_\ p) &\triangleq \texttt{false}\\
\texttt{conflict\_path}\ p'\ (\texttt{Write\_file}\ \_\ \_\ p) &\triangleq \texttt{subpath}\ p'\ p\\
\texttt{conflict\_path}\ p'\ (\texttt{Write\_dir}\ \_\ \_\ p) &\triangleq \texttt{subpath}\ p'\ p\\[4pt]
\texttt{extract\_paths}\ [] &\triangleq \{\}\\
\texttt{extract\_paths}\ ((\texttt{Read}\ \_\ p)\cdot tl) &\triangleq \{p\} \cup (\texttt{extract\_paths}\ tl)\\
\texttt{extract\_paths}\ ((\texttt{Write\_file}\ \_\ \_\ p)\cdot tl) &\triangleq \{p\} \cup (\texttt{extract\_paths}\ tl)\\
\texttt{extract\_paths}\ ((\texttt{Write\_dir}\ \_\ \_\ p)\cdot tl) &\triangleq \{p\} \cup (\texttt{extract\_paths}\ tl)
\end{aligned}$$

Fig. 9. `merge` and `check_log`

Commands are treated as atomic for simplicity they could also be modeled at finer granularity without affecting our results.

With a global semantics where transactions are run concurrently, we now aim to prove that our semantics guarantees serializability. The theorem below captures this property by connecting the operational and denotational semantics:

Theorem 4 (Serializability). *Let FS, FS' be file systems, GL, GL' be global logs, and T a thread pool such that $\forall t \in T.\ \texttt{initial}\ FS\ t$, then:*

$$\langle FS, GL, T\rangle \rightarrow_G^* \langle FS', GL', \{\}\rangle \implies \exists \ell \in Perm(T).\ [\![\ell]\!]_{\mathbb{G}}\ FS = FS'$$

where $\rightarrow_G^*$ is the reflexive, transitive closure of $\rightarrow_G$.

The serializability theorem states that given a starting file system and a thread pool of starting threads, if the global operational semantics commits them all, then there is some ordering of these threads for which the global denotational semantics will produce the same resulting file system. Note that although it is not required by the theorem, the commit order is one such ordering. Additionally, though not explicitly stated, it is easy to see that any serial schedule that is in the domain of the denotation function is realizable by the operational semantics. See the technical report [2] for the proof.

The prototype system described in the next section implements the local semantics from the previous section along with this global semantics, reducing the burden of writing correct concurrent applications.

5 Implementation

This section describes our prototype implementation of Transactional Forest as an embedded domain-specific language in OCaml. Our prototype comprises 6089 lines of code (excluding blank lines and comments) and encodes Forest's features as a PPX syntax extension.

We have implemented a simple course management system similar to the running example from Section 2. It has several additional facilities beyond renormalization, including computing various statistics about students or homeworks and changing rubrics while automatically updating student grades accordingly. The most interesting piece of the example is based on our experience with a professional grading system which uses a queue from which graders can get new problems to grade. Unfortunately, this system did not adequately employ concurrency control, resulting in duplicated work. Using TxForest, we implemented a simple grading queue where graders can add and retrieve problems, which does not suffer from such concurrency issues.

The embedded language in our prototype implementation implements almost precisely the language seen in Section 3. Additionally, we provide a surface syntax (as seen in Section 2 and papers on the earlier versions of Forest [5,3]) for specifications that compiles down to the core calculus seen in Section 3. This specification can then be turned into a zipper by initiating a transaction. The majority of the commands and expressions seen in the core semantics are then exposed as functions in a library. Additionally, there is a more ad hoc surface command language that resembles the surface syntax and parallels the behavior of the core language. Finally, the global semantics looks slightly different compared to in Section 4, though this should not affect users and the minor variant has been proven correct. We provide a simple shell for interacting with filestores, which makes it significantly easier to force conflicts and test the concurrent semantics.

6 Related Work

We discuss four families of related work: languages for ad hoc data processing, zippers, transaction semantics, and transactional file systems.

Ad hoc data processing. Transactional Forest builds on a long line of work in ad hoc data processing. PADS [6,7] (Processing Ad hoc Data Streams) is a declarative domain-specific language designed to deal with ad hoc data. It allows users to write declarative specifications describing the structure of a file and uses such descriptions to generate types, transformations between on-disk and in-memory representations with robust error handling, and various statistical analysis tools.

Forest [5] extends the concept of PADS to full filestores and additionally provides formal guarantees about the generated transformations in the form of bidirectional lens laws. The original version of Forest was implemented in Haskell and relied on its host language's laziness to load only required data. Unfortunately, it was not always clear what actions might trigger loading the entire filestore. For example, checking if there were errors at any level would load everything below that level. Incremental Forest [3] addressed this issue by introducing *delays* to make explicit the amount of loading corresponding to any action. It also supported a cost semantics that precisely characterized the cost of any such action for varied, user-defined notions of cost. It did not address concurrent access to Incremental Forest-described filestores, however.

A bit farther afield, Microsoft's LINQ [12] and F#'s Type Providers [16] share the Forest family's goal of making data-oriented programming easier. While they lack support for declarative specifications of filestores, LINQ and Type Providers both include nice interfaces for interacting with data. In contrast, languages like XFiles [1] do allow declarative filestore specifications, but do not directly interoperate with general purpose programming languages.

Zippers. Huet [10] introduced Zippers as an elegant data structure for traversing and updating a functional tree. There has been much work studying zippers since, though the closest to our work is Kiselyov's Zipper file system [11]. Kiselyov builds a small functional file system with a zipper as its core abstraction. This file system offers a simple transaction mechanism by providing each thread its own view of the file system. The system lacks formal guarantees and is generic: it does not support an application-specific view of the file system as a filestore. In contrast, Transactional Forest uses type-based specifications to describe the structure and invariants of filestores. Further, we present a formal syntax and semantics for our core language, a model of concurrency, and a proof of serializability.

Transaction Semantics. Moore and Grossman [14] present a family of languages with software transactions, investigating how these languages support parallelism and what restrictions are necessary to ensure correctness in the presence of weak isolation. Additionally, they provide a type-and-effect system which ensures the serializability of well-typed programs. At a high level, they describe what the core of a language used to write concurrent programs might look and act like, including constructs like spawning threads or atomic sections. Our transactional semantics is higher-level and specific to our domain, describing a transaction manager designed simply to ensure serializability among TxForest threads.

Transactional File Systems. General support for transactional file systems has been well studied [4,9,13,15]. All of this work starts at a lower level than Transactional Forest, providing transaction support for file system commands. We, instead, provide transactions from the perspective of the higher-level application, easily allowing an arbitrary high-level computation to be aborted or restarted if there is a conflict at the file system level.

7 Conclusion

We have presented the design, syntax, and semantics of Transactional Forest, a domain-specific language for incrementally processing ad hoc data in concurrent applications. TxForest aims to provide an easier and less error-prone approach to modeling and interacting with a structured subset of a file system, which we call a *filestore*. We achieve this by leveraging Huet's Zippers [10] as our core abstraction. Their traversal-based structure naturally lends itself to incrementality and a simple, efficient logging scheme that we use for our optimistic concurrency control. We provide a core language with a formal syntax and semantics based on zipper traversal, both for local, single-threaded applications, and for a global view with arbitrarily many Forest processes. We prove that this global view enforces serializability between threads, that is, the resulting effect on the file system of any set of concurrent threads is the same as if they had run in some serial order. Our OCaml prototype provides a surface language mirroring Classic Forest [5] and a library of functions for manipulating the filestore.

Acknowledgments. The authors wish to thank the anonymous APLAS reviewers for helpful comments on an earlier draft of this paper. Our work is supported in part by the National Science Foundation under grant CNS-1413972 and by the Defense Advanced Research Projects Agency.

References

1. Buraga, S.C.: An XML-based semantic description of distributed file systems. In: RoEduNet (2003)
2. DiLorenzo, J., Mancini, K., Fisher, K., Foster, N.: TxForest: A DSL for Concurrent Filestores. Technical report (2019), https://arxiv.org/abs/1908.10273
3. DiLorenzo, J., Zhang, R., Menzies, E., Fisher, K., Foster, N.: Incremental Forest: a DSL for efficiently managing filestores. In: ACM SIGPLAN Notices. OOPSLA, vol. 51, pp. 252–271 (2016)
4. Escriva, R., Sirer, E.G.: The Design and Implementation of the Warp Transactional Filesystem. In: Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation. pp. 469–483. NSDI'16, USENIX Association, Berkeley, CA, USA (2016)
5. Fisher, K., Foster, N., Walker, D., Zhu, K.Q.: Forest: A Language and Toolkit for Programming with Filestores. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. pp. 292–306. ICFP '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2034773.2034814
6. Fisher, K., Gruber, R.: Pads: A domain-specific language for processing ad hoc data. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 295–304. PLDI '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065046
7. Fisher, K., Walker, D.: The PADS project: An overview. In: Proceedings of the 14th International Conference on Database Theory. ICDT '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1938551.1938556
8. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View Update Problem. ACM Transactions on Programming Languages and Systems (TOPLAS) **29**(3) (May 2007), short version in POPL '05.
9. Garcia, J., Ferreira, P., Guedes, P.: The PerDiS FS: A Transactional File System for a Distributed Persistent Store. In: Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications. pp. 189–194. EW 8, ACM, New York, NY, USA (1998). https://doi.org/10.1145/319195.319224
10. Huet, G.: The Zipper. J. Funct. Program. **7**(5), 549–554 (Sep 1997). https://doi.org/10.1017/S0956796897002864

11. Kiselyov, O.: Tool demonstration: A zipper based file/operating system. In: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell. Haskell '05 (2005), http://okmij.org/ftp/continuations/ZFS/zfs-talk.pdf
12. LINQ: .NET language-integrated query. http://msdn.microsoft.com/library/bb308959.aspx (Feb 2007)
13. Liskov, B., Rodrigues, R.: Transactional File Systems Can Be Fast. In: Proceedings of the 11th Workshop on ACM SIGOPS European Workshop. EW 11, ACM, New York, NY, USA (2004). https://doi.org/10.1145/1133572.1133592
14. Moore, K.F., Grossman, D.: High-level Small-step Operational Semantics for Transactions. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 51–62. POPL, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1328438.1328448
15. Schmuck, F., Wylie, J.: Experience with Transactions in QuickSilver. In: Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles. pp. 239–253. SOSP '91, ACM, New York, NY, USA (1991). https://doi.org/10.1145/121132.121171
16. Syme, D.: Looking ahead with F#: Taming the data deluge (Nov 2010), presentation at the Workshop on F# in Education