TxForest: Composable Memory Transactions over Filestores

Forest Team
Cornell, TUFTS
forest@cs.cornell.edu

Abstract

Keywords

1. Introduction

Databases are a long-standing, effective technology for storing structured and semi-structured data. Using a database has many benefits, including transactions and access to rich set of data manipulation languages and toolkits.

downsides: heavy legacy, relational model is not always adequate

cheaper and simpler alternative: store data directly as a collection of files, directories and symbolic links in a traditional filesystem.

examples of filesystems as databases

filesystems fall short for a number of reasons

Forest [1] made a solid step into solving this, by offering an integrated programming environment for specifying and managing filestores.

Although promising, the old Forest suffered two essential shortcomings:

- It did not offer the level of transparency of a typical DBMS. Users don't get to believe that they are working directly on the database (filesystem). they explicitly issue load/store calls, and instead manipulate in-memory representations and the filesystem independently. offline synchronization.
- It provided none of the transactional guarantees familiar from databases. transactions are nice: prevent concurrency and failure problems. successful transactions are guaranteed to run in serial order and failing transactions rollback as if they never occurred. rely on extra programmers' to avoid the hazards of concurrent updates. different hacks and tricks like creating lock files and storing data in temporary locations, that severely increase the complexity of the applications. writing concurrent programs is notoriously hard to get right. even more in the presence of laziness (original forest used the generally unsound Haskell lazy I/O)

transactional filesystem use cases:

a directory has a group of files that must be processed and deleted and having the aggregate result written to another file.

software upgrade (rollback),

concurrent file access (beautiful account example?)

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm...$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

```
[pads | data Balance = Balance Int |]
[forest |
  type Accounts = [a :: Account | a <- matches (GL "*")]
  type Account = File Balance
 |]
```

```
data Balance = ...
type Balance_md = ...
data Accounts
instance TxForest () Accounts (FileInfo, [(FilePath, Account)]) wher
data Account
instance TxForest () Account ((FileInfo, Balance_md), Balance) wher
```

```
[forest |
  type Universal_d = Directory
    { ascii_files  is [f :: TextFile      | f <- matches (GL "*"), (kind f_a
    , binary_files is [b :: BinaryFile    | b <- matches (GL "*"), (kind b_a
    , directories  is [d :: Universal_d   | d <- matches (GL "*"), (kind d_a
    , symlinks     is [s :: Link          | s <- matches (GL "*"), (isJust (s
    }
 |]
```

Specific use cases: LHC
Network logs
Dan's scientific data

2. Examples

3. The Forest Language

the forest description types

a forest description defines a structured representation of a semi-structured filestore.

each Forest declaraction is interpreted as: an expected on-disk shape of a filesystem fragment a transactional variable an ordinary Haskell type for the in-memory representation that represents the content of a variable

two expression quotations: non-monadic (e) vs monadic $< |e| >$

FileInfo for directories/files/symlinks.

4. Forest Transactions

The Forest description language introduced in the previous section describes how to specify the expected shape of a filestore as an allegorical Haskell type, independently from the concrete programming artifacts that are used to manipulate such filestores. We now focus on the key goal of this paper: the design of the Transactional Forest interface.

As we shall see, TxForest (for short) offers an elegant and powerful abstraction for concurrently manipulating structured filestores. We first describe general-purpose transactional facilities (Section 4.1). We then introduce transactional forest variables that allow programmers to interact with filestores (Section 4.2). We briefly touch on how programmers can verify, at any time, if a filestore conforms to its specification (Section 4.3), and finish by introducing analogues of standard file system operations over filestores (Section 4.4).

4.1 Composable transactions

As an embedded domain-specific language in Haskell, the inspiration for TxForest is the widely popular *software transactional memory* (`STM`) Haskell library, that provides a small set of composable operations to define the key components of a transaction. We now explain the intuition of each one of these mechanisms, cast in the context of TxForest.

Running transactions In TxForest, one runs a transaction by calling the $atomic$ function with type:[1]

$$atomic :: FTM\ a \rightarrow IO\ a$$

It receives a forest memory transaction, of type $FTM\ \ a$, and produces an $IO\ \ a$ action that executes the transaction atomically with respect to all other concurrent transactions, returning a result of type a. In the pure functional language Haskell, FTM and IO are called monads. Different monads are typically used to characterize different classes of computational effects. IO is the primitive Haskell monad for performing irrevocable I/O actions, including reading/writing to files or to mutable references, managing threads, etc. For example, consider the Haskell prelude functions:

$$getChar :: IO\ Char$$
$$putChar :: Char \rightarrow IO\ ()$$

These respectively read a character from the standard input and write a character to the standard output.

Conversely, our FTM monad denotes computations that are tentative, in the sense that they happen inside the scope of a transaction and can always be rolled back. As we discuss in the remainder of this section, these consist of STM-like transactional combinators, file system operations on Forest filestores, or arbitrary pure functions. Note that, since FTM and IO are different types, the Haskell type system effectively prevents non-transactional actions from being run inside of a transaction. This is a valuable guarantee and one that is not commonly found in transactional libraries for mainstream programming languages lacking a very expressive type system.

Blocking transactions To allow a transaction to *block* on a resource, TxForest provides a $retry$ operation with type:

$$retry :: FTM\ a$$

Conceptually, $retry$ cancels the current transaction, without emitting any errors, and schedules it to be retried at a later time. Since each transaction logs all the reads/writes that it performs on a filestore, an efficient implementation waits for another transaction to update the shared filestore fragments read by the blocked transaction before retrying.

Using $retry$ we can define a pattern for conditional transactions that waits on a condition to be verified before performing an action:

$$wait :: FTM\ Bool \rightarrow FTM\ a \rightarrow FTM\ a$$
$$wait\ p\ a = \mathbf{do}\ \{ b \leftarrow p; \mathbf{if}\ b\ \mathbf{then}\ retry\ \mathbf{else}\ a \}$$

Note that $wait$ does not require a cycle; the transactional semantics handles consecutive retries.

Composing transactions Multiple transactions can be sequentially composed via the standard $\mathbf{do}$ notation. For example, we can write:

$$\mathbf{do}\ \{ x \leftarrow ftm1; fmt2\ x \}$$

This runs a transaction $ftm1 : FTM\ \ a$ and passes its result to a transaction $ftm2 :: a \rightarrow FTM\ \ b$. Since the whole computation is itself a transaction, it will be performed indivisibly inside an $atomic$ block.

We can also compose transactions as *alternatives*, using the $orElse$ primitive:

$$orElse :: FTM\ a \rightarrow FTM\ a \rightarrow FTM\ a$$

This combinator performs a left-biased choice: It first runs transaction $ftm1$, tries $ftm2$ if $ftm1$ retries, and the whole transaction retries if $ftm2$ retries. For example, it might be used to read either one of two files depending on the current configuration of the file system.

Note that $orElse$ provides an elegant mechanism for nested transactions. At any point inside a larger transaction, we can tentatively perform a transaction $ftm1$ and rollback to the beginning (of the nested transaction) to try $ftm2$ in case $ftm1$ retries:

$$\mathbf{do}\ \{ ...; orElse\ ftm1\ ftm2; ... \}$$

Exceptions The last general-purpose feature of FTM transactions are *exceptions*. In Haskell, both built-in and user-defined exceptions are used to signal error conditions. We can $throw$ and $catch$ exceptions in the FTM monad in the same way as the IO monad:

$$throw :: Exception\ e \Rightarrow e \rightarrow FTM\ a$$
$$catch :: Exception\ e \Rightarrow FTM\ a \rightarrow (e \rightarrow FTM\ a) \rightarrow FTM\ a$$

For instance, a TxForest user may define a new $FileNotFound$ exception and write the following pseudo-code:

$$\begin{aligned} &tryRead = \mathbf{do} \\ &\quad \{ exists \leftarrow ...find\ file\ ... \\ &\quad ; \mathbf{if}\ (not\ exists)\ \mathbf{then}\ throw\ FileNotFound\ \mathbf{else}\ return\ () \\ &\quad ; ...read\ file... \} \end{aligned}$$

If the file in question is not found, then a $FileNotFound$ exception is thrown, aborting the current $atomic$ block (and hence the file is never read). Programmers can prevent the transaction from being aborted, and its effects discarded, by catching exceptions inside the transaction, e.g.:

$$\begin{aligned} &catch\ tryRead \\ &\quad (\lambda FileNotFound \rightarrow return\ ...default...)\ tryRead \end{aligned}$$

4.2 Transactional variables

We have seen how to build transactions from smaller transactional blocks, but we still haven't seen concrete operations to manipulate *shared data*, a fundamental piece of any transactional mechanism. In vanilla Haskell STM, communication between threads is done via shared mutable memory cells called *transactional variables*. For a transaction to log all memory effects, transactional variables can only be explicitly created, read from or written to using specific

[1] For the original `STM` interface [2], substitute FTM by STM.

transactional operations. Nevertheless, Haskell programmers can traverse, query and manipulate the content of transactional variables using the rich language of purely functional computations; since these don't have side-effects, they don't ever need to be logged or rolled back.

In the context of TxForest, shared data is not stored in-memory, but instead on the filestore. It is illuminating to quote the STM paper [2]:

> "We study internal concurrency between threads interacting through memory [...]; we do not consider here the questions of external interaction through storage systems or databases."

We consider precisely the question of external interaction with a file system. Two transactions may communicate, e.g., by reading from or writing to the same file or possibly a list of files within a directory. To facilitate this interaction, the TxForest compiler generates an instance of the $TxForest$ type class (and corresponding types) for each Forest declaration:

$$
\begin{array}{l}
\textbf{class}\ TxForest\ args\ ty\ rep \mid ty \rightarrow rep, ty \rightarrow args\ \textbf{where}\\
\quad new \quad\quad\ :: args \rightarrow FilePath \rightarrow FTM\ fs\ ty\\
\quad read \quad\quad :: ty \rightarrow FTM\ rep\\
\quad writeOrElse :: ty \rightarrow rep \rightarrow b\\
\quad\quad\quad\quad\quad\quad \rightarrow (Manifest \rightarrow FTM\ fs\ b) \rightarrow FTM\ fs\ b
\end{array}
$$

In this signature, ty is an opaque transactional variable type that uniquely identifies a user-declared Forest type. The representation type rep is a plain Haskell type that holds the content of a transactional variable. The representation type closely follows the declared Forest type, with additional file-content metadata for directories, files and symbolic links; directories have representations of type $(FileInfo, dir\_rep)$ and basic types have representations of type $((FileInfo, base\_md), base\_rep)$, for base representation $base\_rep$ and metadata $base\_md$.

Creation The transactional forest programming style makes no distinction between data on the file system and in-memory. Anywhere inside a transaction, users can declare a new transactional variable, with argument data pertaining to the forest declaration and rooted at the argument path in the file system. This operation does not have any effect on the file system and just establishes the schema to which a filestore should conform.

Reading Users can $read$ data from a filestore by reading the contents of a transactional variable. Imagine that we want to retrieve the balance of a particular account from a directory of accounts as specified in Figure **??**:

$$
\begin{array}{l}
\textbf{do}\\
\quad accs :: Accounts \leftarrow new\ ()\ \texttt{"/var/db/accounts"}\\
\quad (accs\_info, accs\_rep) \leftarrow read\ accs\\
\quad \textbf{let}\ acc1 :: Account = fromJust\ (lookup\ \texttt{"account1"}\ accs\_rep)\\
\quad ((acc1\_info, acc1\_md), Balance\ balance) \leftarrow read\ acc1\\
\quad return\ balance
\end{array}
$$

The corresponding generated Haskell functions and types appear in Figure **??**. In the background, this is done by lazily traversing the directories, files and symbolic links mentioned in the top-level forest description. The second line reads the account directory and generates a list of accounts, which can be manipulated with standard list operations to find the desired account. An account is itself a transactional variable, which can be read in the same way. Note that the file holding the balance of `"account1"` is only read in the fourth line. The type signatures elucidate the type of each transactional variable.

Programmers can control the degree of laziness in a forest description by adjusting the granularity of Forest declarations. For instance, if we have chosen to inline the type of $Account$ in the description as follows:

$$
\begin{array}{l}
[forest \mid\\
\quad \textbf{type}\ Accounts = [\,a :: \texttt{File}\ Balance \mid a \leftarrow matches\ (GL\ \texttt{"*"})]\\
\mid]
\end{array}
$$

Then reading the accounts directory would also read the file content of all accounts, since the balance of each account would not be encapsulated behind a transactional variable (as in Figure **??**.

Writing Users can modify a filestore by writing new content to a transactional variable. The $writeOrElse$ function accepts additional arguments to handle possible conflicts, which may arise due to data dependencies in the Forest description that cannot be statically checked by the type system. If these dependencies are not met, the data is not a valid representation of a filestore. If the write succeeds, the file system is updated with the new data and a default value of type b is returned. If the write fails, a user-supplied alternate function is executed instead. The function takes a $Manifest$ describing the tentative modifications to the file system and a report of the inconsistencies. We can easily define more convenient derived forms of $writeOrElse$:

$$
\begin{array}{l}
\quad \text{-- optional write}\\
tryWrite :: TxForest\ args\ ty\ rep \Rightarrow ty \rightarrow rep \rightarrow FTM\ ()\\
tryWrite\ t\ v = writeOrElse\ t\ v\ ()\ (const\ (return\ ()))\\
\quad \text{-- write or restart the transaction}\\
writeOrRetry :: TxForest\ args\ ty\ rep \Rightarrow ty \rightarrow rep \rightarrow () \rightarrow FTM\ ()\\
writeOrRetry\ t\ v = writeOrElse\ t\ v\ ()\ (const\ retry)\\
\quad \text{-- write or yield an error}\\
writeOrThrow :: (TxForest\ args\ ty\ rep, Exception\ e) \Rightarrow ty \rightarrow rep \rightarrow () \rightarrow\\
writeOrThrow\ t\ v\ e = writeOrElse\ t\ v\ ()\ (const\ (throw\ e))
\end{array}
$$

A typical example of an inconsistent representation is when a Forest description refers to the same file twice and the user attempts to write distinct file content in each occurrence. For instance, in the universal description in Figure **??**, a symbolic link to an ASCII file in the same directory is mapped both under the $ascii\_files$ and $symlinks$ fields.

Writes take immediate effect on the (transactional snapshot of the) filestore, meaning that any subsequent $read$ will see the performed modifications. Within a transaction, there can be multiple variables (possibly of different types) connected to the same fragment of a file system. Consider the following example with two accounts pointing to the same file path:

$$
\begin{array}{l}
acc1 :: Account \leftarrow new\ ()\ \texttt{"/var/db/accounts/account"}\\
acc2 :: Account \leftarrow new\ ()\ \texttt{"/var/db/accounts/account"}\\
(acc\_md, Balance\ balance) \leftarrow read\ acc1\\
tryWrite\ acc2\ (acc\_md, Balance\ (balance + 1))\\
(acc\_md', Balance\ balance') \leftarrow read\ acc1
\end{array}
$$

By incrementing the balance of $acc2$, we are implicitly incrementing the balance of $acc1$ (if the write succeeds, then $balance' = balance + 1$).

4.3 Validation

As Forest lays a structured view on top of a semi-structured file system, a filestore does not need to conform perfectly to an associated Forest description. Behind the scenes, TxForest lazily computes a summary of such discrepancies. These may flag, for example, that a mandatory file does not exist or an arbitrarily complex user-defined Forest constraint is not satisfied. Validation is not performed unless explicitly demanded by the user. At any point, a user can $validate$ a transactional variable and its underlying filestore:

$$
validate :: TxForest\ args\ ty\ rep \Rightarrow ty \rightarrow FTM\ ForestErr
$$

The returned *ForestErr* reports a top-level error count and the topmost error message:

$$\begin{array}{l} \textbf{data}\ \mathit{ForestErr} = \mathit{ForestErr} \\ \quad \{\mathit{numErrors} :: \mathit{Int} \\ \quad , \mathit{errorMsg} \quad :: \mathit{Maybe}\ \mathit{ErrMsg}\} \end{array}$$

We can always make validation mandatory and validation errors fatal by encapsulating any error inside a *ForestError* exception:

$$\begin{array}{l} \mathit{validRead} :: \mathit{TxForest}\ \mathit{args}\ \mathit{ty}\ \mathit{rep} \Rightarrow \mathit{ty} \rightarrow \mathit{FTM}\ \mathit{rep} \\ \mathit{validRead}\ \mathit{ty} = \textbf{do} \\ \quad \mathit{rep} \leftarrow \mathit{read}\ \mathit{ty} \\ \quad \mathit{err} \leftarrow \mathit{validate}\ \mathit{ty} \\ \quad \textbf{if}\ \mathit{numErrors}\ \mathit{err} \equiv 0 \\ \quad\quad \textbf{then}\ \mathit{return}\ \mathit{rep} \\ \quad\quad \textbf{else}\ \mathit{throw}\ (\mathit{ForestError}\ \mathit{err}) \end{array}$$

4.4 Standard file system operations

To better understand the TxForest interface, we now discuss how to perform common operations on a Forest filestore.

Creation/Deletion Given that validation errors are not fatal, a *read* always returns a representation. For example, if a user tries to read the balance of a non-existent account:

$$\begin{array}{l} \textbf{do} \\ \quad \mathit{badAcc} :: \mathit{Account} \leftarrow \mathit{new}\ ()\ \texttt{"/var/db/accounts/account"} \\ \quad (\mathit{acc\_info}, \mathit{Balance}\ \mathit{balance}) \leftarrow \mathit{read}\ \mathit{badAcc} \end{array}$$

then *acc_info* will hold invalid file information and *balance* a default value (implemented as 0 for *Int* values). Perhaps less intuitive is how to create a new account; we create a new variable (that if read would hold default data) and write new valid file information and some balance:

$$\begin{array}{l} \mathit{newAccount}\ \mathit{path}\ \mathit{balance} = \textbf{do} \\ \quad \mathit{newAcc} :: \mathit{Account} \leftarrow \mathit{new}\ ()\ \mathit{path} \\ \quad \mathit{tryWrite}\ \mathit{newAcc}\ (\mathit{validFileInfo}\ \mathit{path}, \mathit{Balance}\ \mathit{balance}) \end{array}$$

Deleting an account is dual to creating one; we write invalid file information and the default balance to the corresponding variable:

$$\begin{array}{l} \mathit{delAccount}\ \mathit{acc} = \textbf{do} \\ \quad \mathit{tryWrite}\ \mathit{acc}\ (\mathit{invalidFile}, \mathit{Balance}\ 0) \end{array}$$

The takeaway lesson is that the *FileInfo* metadata actually determines whether a directory, file or symbolic link exists or not in the file system, since we cannot infer that from the data alone (e.g., an empty account has the same balance as a non-existent account). This also reveals less obvious data dependencies: For valid paths the *fullpath* in the metadata must match the path to which the representation corresponds in the description, and for invalid paths the representation data must match the Forest-generated default data. Since this can become cumbersome to ensure manually, we provide a general function that conveniently removes a filestore, named after the POSIX `rm` operation:

$$\mathit{rm} :: \mathit{TxForest}\ \mathit{args}\ \mathit{ty}\ \mathit{rep} \Rightarrow \mathit{ty} \rightarrow \mathit{FTM}\ ()$$

Copying A user can copy an account from a source path to a target path as follows:

$$\begin{array}{l} \mathit{copyAccount}\ \mathit{srcpath}\ \mathit{tgtpath} = \textbf{do} \\ \quad \mathit{src} :: \mathit{Account} \leftarrow \mathit{new}\ ()\ \mathit{srcpath} \\ \quad \mathit{tgt} :: \mathit{Account} \leftarrow \mathit{new}\ ()\ \mathit{tgtpath} \\ \quad (\mathit{info}, \mathit{balance}) \leftarrow \mathit{read}\ \mathit{src} \\ \quad \mathit{tryWrite}\ \mathit{tgt}\ (\mathit{info}\ \{\mathit{fullpath} = \mathit{tgtpath}\}, \mathit{balance}) \end{array}$$

The pattern is to create a variable for each path, and copy the content with an updated *fullpath*. Copying a directory of accounts follows the same pattern but is more complicated, in that we also have to recursively copy underlying accounts and update all the metadata accordingly. Therefore, we provide an analogue to the POSIX `cp` operation that attempts to copy the content of a representation into another:

$$\begin{array}{l} \mathit{cpOrElse} :: \mathit{TxForest}\ \mathit{args}\ \mathit{ty}\ \mathit{rep} \Rightarrow \mathit{ty} \rightarrow \mathit{ty} \rightarrow b \\ \qquad\qquad \rightarrow (\mathit{Manifest} \rightarrow \mathit{rep} \rightarrow \mathit{FTM}\ \mathit{fs}\ b) \rightarrow \mathit{FTM}\ \mathit{fs}\ b \end{array}$$

Unlike *rm*, *copyOrElse* is only a best-effort operation that may fail due to arbitrarily complex data dependencies in the Forest description. Such dependencies necessarily hold in the source representation for the source arguments but may not for the target arguments. Similarly to *writeOrElse*, we provide *tryCopy*, *copyOrRetry* and *copyOrThrow* operations with the expected type signatures.

For an example of what might go wrong while copying, consider the following description for accounts parameterized by a template name:

$$\begin{array}{l} [\mathit{forest}\ | \\ \quad \textbf{type}\ \mathit{NamedAccounts}\ (\mathit{acc} :: \mathit{String}) = \\ \quad\quad [a :: \mathit{Account} \mid a \leftarrow \mathit{matches}\ (\mathit{GL}\ (\mathit{acc} +\!\!+ \texttt{"*"}))] \\ |] \end{array}$$

This specification has an implicit data dependency that all the account files listed in the in-memory representation have names matching the Glob pattern. Thus, trying to copy between filestores with different templates would effectively fail, as in:

$$\begin{array}{l} \textbf{do} \\ \quad \mathit{src} :: \mathit{Accounts} \leftarrow \mathit{new}\ \texttt{"account"}\ \texttt{"/var/db/accounts"} \\ \quad \mathit{tgt} :: \mathit{Accounts} \leftarrow \mathit{new}\ \texttt{"acc"}\ \texttt{"/var/db/accs"} \\ \quad \mathit{tryCopy}\ \mathit{src}\ \mathit{tgt} \end{array}$$

5. Implementation

We now delve into how Transactional Forest can be efficiently implemented. The current implementation is available from the project website (`forestproj.org`) and is done completely in Haskell.

increasing levels of incremental support, and added complexity.

5.1 Transactional Forest

optimistic concurrency control

(this is important since we write to canonical paths, whose canonicalization may depend on concurrent writes...)

lock-free lazy acquire acquire ownership. only one tx can acquire an object at a time. global total order on variables, acquire variables in sorted order the analogous in txforest would be per-filepath locks, what does nto work out-of-the-box in the presence of symbolic links

the identity of a filepath is not unique (different paths point to the same physical address) nor stable (equivalence depends on on the current filesystem).

transactional semantics of STM: we log reads/writes to the filesystem instead of variables. global lock, no equality check on validation. load/store semantics of Forest with thunks, explicit laziness

transactional variables created by calling load on its spec with given arguments and root path; lazy loading, so no actual reads occur. Additionally to the representation data, each transactional variable remembers its creation-time arguments (they never change).

each transaction keeps a local filesystem version number, and a per-tvar log mapping fsversions to values, stored in a weaktable (fsversions are purgeable once a tx commits).

on writes: backup the current fslog, increment the fsversion, add an entry to the table for the (newfsversion,newvalue), run the store function for the new data and writing the modifications to the buffered FS; if there are errors, rollback to the backed-up FS and the previous fsversion.

the store function also changes the in-memory representation by recomputing the validation thunks (hidden to users) to match the new content.

write success theorem: if the current rep is in the image of load, then store succeeds

5.2 Incremental Transactional Forest

problem with 1st approach: ic loading: two variables over the same file; read spec1, write spec2, read spec1 (our simple cache mechanism fails to prevent recomputation) laziness problem with 1st approach: ic storing: read variable (child variables are lazy), write variable (will recursively store everything); instead of no-op!

exploit DSL information to have incrementality

5.3 Log-structured Transactional Forest

problem with 2nd approach: tx1 reads a variable; tx2 reads the same variable

exploit (DSL info +) FS support to have incrementality

read-only transactions require no synchronization

6. Evaluation

although Haskell is a great language laboratory, we are already paying a severe performance overhead if efficiency is the only concern.

even the Haskell STM is implemented in C

7. Related Work

transactional filesystems (user-space vs kernel-space) `http://www.fuzzy.cz/en/articles/transactional-file-systems`
`http://www.fsl.cs.sunysb.edu/docs/valor/valor_fast2009.pdf`
`http://www.fsl.cs.sunysb.edu/docs/amino-tos06/amino.pdf`

libraries for transactional file operations: `http://commons.apache.org/proper/commons-transaction/file/index.html`
`https://xadisk.java.net/`
`https://transactionalfilemgr.codeplex.com/`

tx file-level operations (copy,create,delete,move,write) schema somehow equivalent to using the unstructured universal Forest representation

but what about data manipulation: transactional maps,etc?

8. Conclusions

transactional variables do not descend to the content of files. pads specs are read/written in bulk. e.g., append line to log file. extend pads.

References

[1] K. Fisher, N. Foster, D. Walker, and K. Q. Zhu. Forest: A language and toolkit for programming with filestores. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 292–306. ACM, 2011.

[2] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. . URL `http://doi.acm.org/10.1145/1065944.1065952`.

A. Forest Semantics

$$F^*(r\ /\ u) = \begin{cases} F^*(r') & \text{if } F(F^*(r)\ /\ u) = (i, \texttt{Link}\ r') \\ F^*(r)\ /\ u & \text{otherwise} \end{cases}$$
$$F^*(\cdot) = \cdot$$

$$\frac{}{r \in \cdot} \qquad \frac{}{r \in r} \qquad \frac{r \in r'}{r\ /\ u \in r'}$$

$$F{\searrow}r \triangleq F|_{\{\forall\, r'.\ F^*(r') \in r\}}$$

$$r_1 \in^*_F r_2 = \forall\, r \in r_1.\ F^*(r) \in F^*(r_2)$$

$$F \underset{rs}{=} F' = \forall\, r \in rs.\ F{\searrow}r = F'{\searrow}r$$

$$\mathit{Err}\ a = (\mathsf{M}\ \mathit{Bool}, a)$$

s	$\mathcal{R}[\![s]\!]$	$\mathcal{C}[\![s]\!]$
$\mathsf{M}\ s$	$\mathsf{M}\ (\mathit{Err}\ (\mathcal{R}[\![s]\!]))$	$\mathsf{M}\ (\mathit{Err}\ (\mathcal{R}[\![s]\!]))$
$k^{\tau_2}_{\tau_1}$	$\mathit{Err}\ (\tau_2, \tau_1)$	(τ_2, τ_1)
$e :: s$	$\mathcal{R}[\![s]\!]$	$\mathcal{C}[\![s]\!]$
$\langle x : s_1, s_2 \rangle$	$\mathit{Err}\ (\mathcal{R}[\![s_1]\!], \mathcal{R}[\![s_2]\!])$	$(\mathcal{C}[\![s_1]\!], \mathcal{C}[\![s_2]\!])$
$\{s \mid x \in e\}$	$\mathit{Err}\ [\mathcal{R}[\![s]\!]]$	$[\mathcal{C}[\![s]\!]]$
$\mathsf{P}\ (e)$	$\mathit{Err}\ ()$	$()$
$s?$	$\mathit{Err}\ (\mathit{Maybe}\ (\mathcal{R}[\![s]\!]))$	$\mathit{Maybe}\ (\mathcal{C}[\![s]\!])$

$\mathcal{R}[\![\cdot]\!]$ is the internal in-memory representation type of a forest declaration; $\mathcal{C}[\![\cdot]\!]$ is the external type of content of a variables that users can inspect/modify

$$\mathit{err}(a) = \texttt{do}\ \{e \leftarrow \texttt{get}\ a; (a_{err}, v) \leftarrow e; \texttt{return}\ a_{err}\}$$
$$\mathit{err}(a_{err}, v) = \texttt{return}\ a_{err}$$
$$\mathit{valid}(v) = \texttt{do}\ \{a_{err} \leftarrow \mathit{err}\ v; e_{err} \leftarrow \texttt{get}\ a_{err}; e_{err}\}$$

$v_1\ {}_{\Theta_1}{\sim}_{\Theta_2}\ v_2$ denotes value equivalence modulo memory addresses, under the given environments. $e_1\ {}_{\Theta_1}{\sim}_{\Theta_2}\ e_2$ denotes expression equivalence by evaluation modulo memory addresses, under the given environments.

$v_1\ {}_{\Theta_1}\overset{\text{err}}{\sim}_{\Theta_2}\ v_2$ denotes value equivalence (ignoring error information) modulo memory addresses, under the given environments.

$\boxed{\Theta; \varepsilon; r; s \vdash \texttt{load}\ F \Rightarrow \Theta'; v}$ "Under heap Θ and environment ε, load the specification s for filesystem F at path r and yield a representation v."

$\boxed{s = \mathsf{M}\ s_1}$

$$\frac{a \notin \texttt{dom}(\Theta) \quad a_{err} \notin \texttt{dom}(\Theta) \quad e = \varepsilon; r; \mathsf{M}\ s \vdash \texttt{load}\ F \\ e_{err} = \texttt{do}\ \{e_1 \leftarrow \texttt{get}\ a; v_1 \leftarrow e_1; \mathit{valid}\ v_1\}}{\Theta; \varepsilon; r; \mathsf{M}\ s \vdash \texttt{load}\ F \Rightarrow \Theta[a_{err} : e_{err}, a : e]; (a_{err}, a)}$$

$\boxed{s = k}$

$$\frac{a_{err} \notin \texttt{dom}(\Theta) \quad \Theta; \texttt{load}_k(\varepsilon, F, r) \Rightarrow \Theta'; (b, v)}{\Theta; \varepsilon; r; k \vdash \texttt{load}\ F \Rightarrow \Theta'[a_{err} : \texttt{return}\ b]; (a_{err}, v)}$$

$$\texttt{load}_{\texttt{File}}(\varepsilon, F, r) \begin{cases} \texttt{return}\ (\mathit{True}, (i, u)) & \text{if } F(r) = (i, \texttt{File}\ u) \\ \texttt{return}\ (\mathit{False}, (i_{\texttt{invalid}}, \texttt{""})) & \text{otherwise} \end{cases}$$

$$\texttt{load}_{\texttt{Dir}}(\varepsilon, F, r) \begin{cases} \texttt{return}\ (\mathit{True}, (i, us)) & \text{if } F(r) = (i, \texttt{Dir}\ us) \\ \texttt{return}\ (\mathit{False}, (i_{\texttt{invalid}}, \{\,\})) & \text{otherwise} \end{cases}$$

$$\texttt{load}_{\texttt{Link}}(\varepsilon, F, r) \begin{cases} \texttt{return}\ (\mathit{True}, (i, r')) & \text{if } F(r) = (i, \texttt{Link}\ r') \\ \texttt{return}\ (\mathit{False}, (i_{\texttt{invalid}}, \cdot)) & \text{otherwise} \end{cases}$$

$\boxed{s = e :: s_1}$

$$\frac{\Theta; [\![r\ /\ e]\!]^{\varepsilon}_{\mathit{Path}} \Rightarrow \Theta'; r' \quad \Theta; \varepsilon; r'; s \vdash \texttt{load}\ F \Rightarrow \Theta''; v}{\Theta; \varepsilon; r; e :: s \vdash \texttt{load}\ F \Rightarrow \Theta''; v}$$

$\boxed{s = \langle x : s_1, s_2 \rangle}$

$$\frac{\begin{array}{c}\Theta; \varepsilon; r; s_1 \vdash \texttt{load}\ F \Rightarrow \Theta_1; v_1 \\ \Theta_1; \varepsilon[x \mapsto v_1]; r; s_2 \vdash \texttt{load}\ F \Rightarrow \Theta_2; v_2 \\ e_{err} = \texttt{do}\ \{\, b_1 \leftarrow \mathit{valid}(v_1);\ b_2 \leftarrow \mathit{valid}(v_2); \texttt{return}\ (b_1 \wedge b_2)\,\}\end{array}}{\Theta; \varepsilon; r; \langle x : s_1, s_2 \rangle \vdash \texttt{load}\ F \Rightarrow \Theta_2[a_{err} : e_{err}]; (a_{err}, (v_1, v_2))}$$

$\boxed{s = \mathsf{P}\ e}$

$$\frac{a_{err} \notin \texttt{dom}(\Theta)}{\Theta; \varepsilon; r; \mathsf{P}\ e \vdash \texttt{load}\ F \Rightarrow \Theta[a_{err} : [\![e]\!]^{\varepsilon}_{Bool}]; (a_{err}, ())}$$

$\boxed{s = s_1?}$

$$\frac{r \notin \texttt{dom}(F) \quad a_{err} \notin \texttt{dom}(\Theta)}{\Theta; \varepsilon; r; s? \vdash \texttt{load}\ F \Rightarrow \Theta[a_{err} : \texttt{return}\ \mathit{True}]; (a_{err}, \mathit{Nothing})}$$

$$\frac{r \in \texttt{dom}(F) \quad a_{err} \notin \texttt{dom}(\Theta') \quad \Theta; \varepsilon; r; s \vdash \texttt{load}\ F \Rightarrow \Theta'; v}{\Theta; \varepsilon; r; s? \vdash \texttt{load}\ F \Rightarrow \Theta[a_{err} : \mathit{valid}(v)]; (a_{err}, \mathit{Just}\ v)}$$

$\boxed{s = \{s_1 \mid x \in e\}}$

$$\frac{\begin{array}{c} a_{err} \notin \texttt{dom}(\Theta) \quad \Theta; [\![e]\!]^{\varepsilon}_{\{\tau\}} \Rightarrow \Theta'; \{\, t_1, .., t_k \,\} \\ \Theta'; \forall\, i \in \{1, ..., k\}.\ \texttt{do}\ \{ v_i \leftarrow \varepsilon[x \mapsto t_i]; r; s \vdash \texttt{load}\ F; \texttt{return}\ \{t_i \mapsto v_i\}\} \Rightarrow \Theta''; \mathit{vs} \\ e_{err} = \forall\, i \in \{1, ..., k\}.\ \texttt{do}\ \{ b_i \leftarrow \mathit{valid}(\mathit{vs}(t_i)); \texttt{return}\ (\bigwedge\ b_i)\}\end{array}}{\Theta; \varepsilon; r; \{s \mid x \in e\} \vdash \texttt{load}\ F \Rightarrow \Theta''[a_{err} : e_{err}]; (a_{err}, \mathit{vs})}$$

$\boxed{\Theta; \varepsilon; r; s \vdash \texttt{store}\ F\ v \Rightarrow \Theta'; (F', \phi')}$ "Under heap Θ and environment ε, store the representation v for the specification s on filesystem F at path r and yield an updated filesystem F' and a validation function ϕ'."

$\boxed{s = \mathsf{M}\, s_1}$

$$\frac{\begin{array}{c}\Theta(a) = e \quad \Theta; e \Rightarrow \Theta'; (a_{err}, v) \\ \Theta'; \varepsilon; r; s \vdash \texttt{store}\ F\ v \Rightarrow \Theta''; (F', \phi')\end{array}}{\Theta; \varepsilon; r; \mathsf{M}\, s \vdash \texttt{store}\ F\ a \Rightarrow \Theta''; (F', \phi')}$$

$\boxed{s = k}$

$$\frac{\Theta; \texttt{store}_k(\varepsilon, F, r, (d, v)) \Rightarrow \Theta'; (F', \phi)}{\Theta; \varepsilon; r; k \vdash \texttt{store}\ F\ (a_{err}, (d, v)) \Rightarrow \Theta'; (F', \phi)}$$

$$\texttt{store}_{\texttt{File}}(\varepsilon, F, r, (i, u)) \begin{cases} \texttt{return}\ (F[r := (i, \texttt{File}\ u)], \lambda F'.\ F'(r) = (i, \texttt{File}\ u)) & \text{if } i \neq i_{\texttt{invalid}} \\ \texttt{return}\ (F[r := \bot], \lambda F'.\ F'(r) \neq (\_, \texttt{File}\ \_)) & \text{if } i = i_{\texttt{invalid}} \wedge F(r) = (\_, \texttt{File}\ \_) \\ \texttt{return}\ (F, \lambda F'.\ F'(r) \neq (\_, \texttt{File}\ \_)) & \text{if } i = i_{\texttt{invalid}} \wedge F(r) \neq (\_, \texttt{File}\ \_) \end{cases}$$

$$\texttt{store}_{\texttt{Dir}}(\varepsilon, F, r, (i, \{\, u_1, ..., u_n \,\})) \begin{cases} \texttt{return}\ (F[r := (i, \texttt{Dir}\ \{\, u_1, ..., u_n \,\})], \lambda F'.\ F'(r) = (i, \texttt{Dir}\ \{\, u_1, ..., u_n \,\})) & \text{if } i \neq i_{\texttt{invalid}} \\ \texttt{return}\ (F[r := \bot], \lambda F'.\ F'(r) \neq (\_, \texttt{Dir}\ \_)) & \text{if } i = i_{\texttt{invalid}} \wedge F(r) = (\_, \texttt{Di} \\ \texttt{return}\ (F, \lambda F'.\ F'(r) \neq (\_, \texttt{Dir}\ \_)) & \text{if } i = i_{\texttt{invalid}} \wedge F(r) \neq (\_, \texttt{Di} \end{cases}$$

$$\texttt{store}_{\texttt{Link}}(\varepsilon, F, r, (i, r')) \begin{cases} \texttt{return}\ (F[r := (i, \texttt{Link}\ r')], \lambda F'.\ F'(r) = (i, \texttt{Link}\ r')) & \text{if } i \neq i_{\texttt{invalid}} \\ \texttt{return}\ (F[r := \bot], \lambda F'.\ F'(r) \neq (\_, \texttt{Link}\ \_)) & \text{if } i = i_{\texttt{invalid}} \wedge F(r) = (\_, \texttt{Link}\ \_) \\ \texttt{return}\ (F, \lambda F'.\ F'(r) \neq (\_, \texttt{Link}\ \_)) & \text{if } i = i_{\texttt{invalid}} \wedge F(r) \neq (\_, \texttt{Link}\ \_) \end{cases}$$

$\boxed{s = e :: s_1}$

$$\frac{\begin{array}{c}\Theta; [\![e]\!]^{\varepsilon}_{Path} \Rightarrow \Theta'; r' \\ \Theta'; \varepsilon; r'; s \vdash \texttt{store}\ F\ v \Rightarrow \Theta''; (F', \phi')\end{array}}{\Theta; \varepsilon; r; e :: s \vdash \texttt{store}\ F\ v \Rightarrow \Theta''; (F', \phi')}$$

$\boxed{s = \langle x : s_1, s_2 \rangle}$

$$\frac{\begin{array}{c}\Theta; \varepsilon; r; s_1 \vdash \texttt{store}\ F\ v_1 \Rightarrow \Theta_1; (F_1, \phi_1) \\ \Theta_1; \varepsilon[x \mapsto v_1]; r; s_2 \vdash \texttt{store}\ F\ v_2 \Rightarrow \Theta_2; (F_2, \phi_2) \\ \phi = \lambda F'.\ \phi_1(F') \wedge \phi_2(F')\end{array}}{\Theta; \varepsilon; r; \langle x : s_1, s_2 \rangle \vdash \texttt{store}\ F\ (a_{err}, (v_1, v_2)) \Rightarrow \Theta_2; (F_1 +\!\!+ F_2, \phi)}$$

$\boxed{s = \mathsf{P}\ e}$

$$\frac{\phi = \lambda F'.\ \mathit{True}}{\Theta; \varepsilon; r; \mathtt{P}\ e \vdash \mathtt{store}\ F\ (a_{err}, ()) \Rightarrow \Theta; (F, \phi)}$$

$\boxed{s = s_1?}$

$$\frac{\phi = \lambda F'.\ r \notin \mathtt{dom}(F')}{\Theta; \varepsilon; r; s? \vdash \mathtt{store}\ F\ (a_{err}, \mathit{Nothing}) \Rightarrow \Theta; (F[r := \bot], \phi)}$$

$$\frac{\begin{array}{c}\Theta; \varepsilon; r; s \vdash \mathtt{store}\ F\ v \Rightarrow \Theta'; (F_1, \phi_1)\\ \phi = \lambda F'.\ \phi_1(F') \wedge r \in \mathtt{dom}(F')\end{array}}{\Theta; \varepsilon; r; s? \vdash \mathtt{store}\ F\ (a_{err}, \mathit{Just}\ v) \Rightarrow \Theta; (F_1, \phi)}$$

$\boxed{s = \{ s_1 \mid x \in e \}}$

$$\frac{\begin{array}{c}\Theta; [\![e]\!]^{\varepsilon}_{\{\tau\}} \Rightarrow \Theta'; ts \quad vs = \{ t_1 \mapsto v_1, ..., t_k \mapsto v_k \}\\ \phi = \lambda F'.\ ts = \{ t_1, ..., t_k \} \wedge \bigwedge\ \phi_i(F')\\ \Theta'; \forall\, i \in \{1, ..., k\}.\ \mathtt{do}\ \{ (F_i, \phi_i) \leftarrow \varepsilon[x \mapsto v_i]; r; s \vdash \mathtt{store}\ F\ v_i; \mathtt{return}\ (F_1 +\!\!\!+ \ ... +\!\!\!+\ F_k, \phi) \} \Rightarrow \Theta''; F'\ \phi'\end{array}}{\Theta; \varepsilon; r; \{ s \mid x \in e \} \vdash \mathtt{store}\ F\ (a_{err}, vs) \Rightarrow \Theta; (F', \phi')}$$

Proposition 1 (Load Type Safety)**.** *If* $\Theta; \varepsilon; r; s \vdash \mathtt{load}\ F \Rightarrow \Theta'; v'$ *and* $\mathcal{R}[\![s]\!] = \tau$ *then* $\vdash v : \tau$.

Theorem A.1 (LoadStore)**.** *If*

$$\begin{aligned} \Theta; \varepsilon; r; s \vdash \mathtt{load}\ F &\Rightarrow \Theta'; v\\ \Theta''; \varepsilon; r; s \vdash \mathtt{store}\ F\ v' &\Rightarrow \Theta'''; (F', \phi')\\ v\ {}_{\Theta'}\!\overset{err}{\sim}_{\Theta''}\ v' \end{aligned}$$

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

Theorem A.2 (StoreLoad)**.** *If*

$$\begin{aligned} \Theta; \varepsilon; r; s \vdash \mathtt{store}\ F\ v &\Rightarrow \Theta'; (F', \phi')\\ \Theta'; \varepsilon; r; s \vdash \mathtt{load}\ F &\Rightarrow \Theta''; v' \end{aligned}$$

then $\phi'(F')$ *iff* $v\ {}_{\Theta'}\!\overset{err}{\sim}_{\Theta''}\ v'$

stronger than the original forest theorem: store validation only fails for impossible cases (when representation cannot be stored to the FS without loss)

weaker in that we don't track consistency of inner validation variables; equality of the values is modulo error information. in a real implementation we want to repair error information on storing, so that it is consistent with a subsequent load.

the error information is not stored back to the FS, so the validity predicate ignores it.

B. Forest Incremental Semantics

Note that:

- We have access to the old filelesystem, since filesystem deltas record the changes to be performed.
- We do not have access to the old environment, since variable deltas record the changes that already occurred.

$\delta_F ::= \mathtt{addFile}(r, u) \mid \mathtt{addDir}(r) \mid \mathtt{addLink}(r, r') \mid \mathtt{rem}(r) \mid \mathtt{chgAttrs}(r, i) \mid \delta_{F_1}; \delta_{F_2} \mid \emptyset$

$\delta_v ::= \mathsf{M}_{\delta_a}\ \delta_{v_1} \mid \delta_{v_1} \otimes \delta_{v_2} \mid \{ t_i \mapsto \delta_{\bot v_i} \} \mid \delta_{v_1}? \mid \emptyset \mid \Delta$
$\delta_{\bot v} ::= \bot \mid \delta_v$

$\Delta_v ::= \emptyset \mid \Delta$

$\mathtt{addFile}(r', u) \searrow_F r \triangleq$ **if** $r' \in^*_F r$ **then** $\mathtt{addFile}(r', u)$ **else** $\emptyset$
$\mathtt{addDir}(r') \searrow_F r \triangleq$ **if** $r' \in^*_F r$ **then** $\mathtt{addDir}(r')$ **else** $\emptyset$
$\mathtt{addLink}(r', r'') \searrow_F r \triangleq$ **if** $r' \in^*_F r$ **then** $\mathtt{addLink}(r', r'')$ **else** $\emptyset$
$\mathtt{rem}(r') \searrow_F r \triangleq$ **if** $r' \in^*_F r$ **then** $\mathtt{rem}(r')$ **else** $\emptyset$
$\mathtt{chgAttrs}(r', i) \searrow_F r \triangleq$ **if** $r' \in^*_F r$ **then** $\mathtt{chgAttrs}(r', i)$ **else** $\emptyset$
$(\delta_{F_1}; \delta_{F_2}) \searrow_F r \triangleq \delta_{F_1} \searrow_F r; \delta_{F_2} \searrow_{F_1} r$ **where** $F_1 = (\delta_{F_1} \searrow_F r)\ F$
$\emptyset \searrow_F r \triangleq \emptyset$

$\Theta; v \xrightarrow{\delta_v} \Theta'; v'$

the value delta maps v to v'

monadic expressions only read from the store and perform new allocations; they can't modify existing addresses.

For any expression application $e\ \Theta = (\Theta', v)$, we have $\Theta = \Theta \cap \Theta'$.

errors are computed in the background

$$\frac{a' \notin \mathtt{dom}(\Theta)}{\Theta; \delta_a; \Delta_e \vdash a : e \Rightarrow \Theta[a' : e]; (a', \Delta)} \quad \frac{}{\Theta; \emptyset; \Delta_e \vdash a : e \Rightarrow \Theta[a : e]; (a, \Delta)} \quad \frac{}{\Theta; \emptyset; \emptyset \vdash a : e \Rightarrow \Theta; (a, \emptyset)}$$

$\boxed{\Theta; \varepsilon; \Delta_\varepsilon; r; s \vdash \mathtt{load}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta'; (v', \Delta'_v)}$ "Under heap Θ, environment ε and delta environment Δ_ε, incrementally load the specification s for the original filesystem F and original representation v, given filesystem changes δ_F and representation changes δ_v, to yield an updated representation v' with changes Δ'_v.

$$\frac{\Delta_\varepsilon|_{fv(s)} = \emptyset \quad \delta_F \searrow_F r = \emptyset}{\Theta; \varepsilon; \Delta_\varepsilon; r; s \vdash \mathtt{load}_\Delta\ F\ v\ \delta_F\ \emptyset \Rightarrow \Theta; (v, \emptyset)}$$

$$\frac{\Theta; \varepsilon; r; s \vdash \mathtt{load}\ (\delta_F\ F) \Rightarrow \Theta'; v'}{\Theta; \varepsilon; \Delta_\varepsilon; r; s \vdash \mathtt{load}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta'; (v', \Delta)}$$

$\boxed{s = \mathsf{M}\, s_1}$

$$\frac{\Theta(a) = e \quad \Theta; e \Rightarrow \Theta'; (a_{err}, v) \qquad \Theta'; \varepsilon; \Delta_\varepsilon; r; s \vdash \mathtt{load}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta''; (v', \Delta_v) \quad v = v'}{\Theta; \varepsilon; \Delta_\varepsilon; r; \mathsf{M}\, s \vdash \mathtt{load}_\Delta\ F\ a\ \delta_F\ (\mathsf{M}_\emptyset\ (\emptyset \otimes \delta_v)) \Rightarrow \Theta''; (a, \emptyset)}$$

$$\frac{\begin{array}{c}\Theta(a) = e \quad \Theta; e \Rightarrow \Theta'; (a_{err}, v) \\ \Theta'; \varepsilon; \Delta_\varepsilon; r; s \vdash \mathtt{load}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta_1; (v', \Delta_v) \\ \Theta_1; \delta_{a_{err}}; \Delta_v \vdash a_{err} : \mathit{valid}\ v' \Rightarrow \Theta_2; (a'_{err}, \Delta_{a_{err}}) \\ \Theta_2; \delta_a; \Delta_{a_{err}} \vdash a : \mathtt{return}\ (a'_{err}, v') \Rightarrow \Theta_3; (a', \Delta_a)\end{array}}{\Theta; \varepsilon; \Delta_\varepsilon; r; \mathsf{M}\, s \vdash \mathtt{load}_\Delta\ F\ a\ \delta_F\ (\mathsf{M}_{\delta_a}\ (\delta_{a_{err}} \otimes \delta_v)) \Rightarrow \Theta_3; (a', \Delta_a)}$$

$\boxed{s = e :: s_1}$

$$\frac{\begin{array}{c}\Delta_\varepsilon|_{fv(s)} = \emptyset \quad \Theta; [\![r\ /\ e]\!]^{\varepsilon}_{Path} \Rightarrow \Theta'; r' \\ \Theta'; \varepsilon; \Delta_\varepsilon; r'; e :: s \vdash \mathtt{load}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta''; (v', \Delta_v)\end{array}}{\Theta; \varepsilon; \Delta_\varepsilon; r; e :: s \vdash \mathtt{load}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta''; (v', \Delta_v)}$$

$\boxed{s = \langle x : s_1, s_2 \rangle}$

$$\frac{\begin{array}{c}\Theta; \varepsilon; \Delta_\varepsilon; r; s_1 \vdash \mathtt{load}_\Delta\ F\ v_1\ \delta_F\ \delta_{v_1} \Rightarrow \Theta_1; (v'_1, \Delta_{v_1}) \\ \Theta_1; \varepsilon[x \mapsto v'_1]; \Delta_\varepsilon[x \mapsto \Delta_{v_1}]; r; s_2 \vdash \mathtt{load}_\Delta\ F\ v_2\ \delta_F\ \delta_{v_2} \Rightarrow \Theta_2; (v'_2, \Delta_{v_2}) \\ \Theta_2; \delta_{a_{err}}; (\Delta_{v_1} \wedge \Delta_{v_2}) \vdash a_{err} : \mathtt{do}\ \{\, b_1 \leftarrow \mathit{valid}\ v'_1; b_2 \leftarrow \mathit{valid}\ v'_2; \mathtt{return}\ (b_1 \wedge b_2) \,\} \Rightarrow \Theta'; (a'_{err}, \Delta_{a_{err}})\end{array}}{\Theta; \varepsilon; \Delta_\varepsilon; r; \langle x : s_1, s_2 \rangle \vdash \mathtt{load}_\Delta\ F\ (a_{err}, (v_1, v_2))\ \delta_F\ (\delta_{a_{err}} \otimes (\delta_{v_1} \otimes \delta_{v_2})) \Rightarrow \Theta'; ((a'_{err}, (v'_1, v'_2)), \Delta_{a_{err}})}$$

$\boxed{s = \mathsf{P}\, e}$

$$\frac{\Delta_\varepsilon|_{fv(e)} = \emptyset}{\Theta; \varepsilon; \Delta_\varepsilon; r; \mathsf{P}\, e \vdash \mathtt{load}_\Delta\ F\ v\ \delta_F\ \emptyset \Rightarrow \Theta; (v, \emptyset)}$$

$\boxed{s = s_1?}$

$$\frac{r \notin \mathtt{dom}(\delta_F\ F) \quad \Theta; \delta_{a_{err}}; \delta_v \vdash a_{err} : \mathtt{return}\ \mathit{True} \Rightarrow \Theta'; (a'_{err}, \Delta_{a_{err}})}{\Theta; \varepsilon; \Delta_\varepsilon; r; s? \vdash \mathtt{load}_\Delta\ F\ (a_{err}, \mathit{Nothing})\ \delta_F\ (\delta_{a_{err}} \otimes \delta_v) \Rightarrow \Theta'; ((a_{err}, \mathit{Nothing}), \Delta_{a_{err}})}$$

$$\frac{\begin{array}{c}r \in \mathtt{dom}(\delta_F\ F) \quad \Theta; \varepsilon; \Delta_\varepsilon; r; s \vdash \mathtt{load}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta'; (v', \Delta_v) \\ \Theta; \delta_{a_{err}}; \Delta_v \vdash a_{err} : \mathit{valid}(v') \Rightarrow \Theta'; (a'_{err}, \Delta_{a_{err}})\end{array}}{\Theta; \varepsilon; \Delta_\varepsilon; r; s? \vdash \mathtt{load}_\Delta\ F\ (a_{err}, \mathit{Just}\ v)\ \delta_F\ (\delta_{a_{err}} \otimes \delta_v?) \Rightarrow \Theta'; ((a_{err}, \mathit{Just}\ v'), \Delta_{a_{err}})}$$

$\boxed{s = \{ s \mid x \in e \}}$

$$\frac{\begin{array}{c}\Theta; [\![e]\!]^{\varepsilon}_{\{\tau\}} \Rightarrow \Theta_1; \{ t_1, ..., t_k \} \\ \Theta_1; \forall\, i \in \{1, ..., k\}.\ \mathtt{do}\ \{ (v_i, \Delta_{v_i}) \leftarrow \varepsilon; \Delta_\varepsilon; r; s \vdash_x \mathtt{load}_\Delta\ F\ vs\ \delta_F\ \delta_{vs}; \mathtt{return}\ (\{ t_i \mapsto v_i \}, \bigwedge\ \Delta_{v_i}) \} \Rightarrow \Theta_2; (vs', \Delta_{vs}) \\ \Theta_2; \delta_{a_{err}}; \Delta_{vs} \vdash a_{err} : \forall\, i \in \{1, ..., k\}.\ \mathtt{do}\ \{ b_i \leftarrow \mathit{valid}(vs'(t_i)); \mathtt{return}\ (\bigwedge\ b_i) \} \Rightarrow \Theta'; (a'_{err}, \Delta_{a_{err}})\end{array}}{\Theta; \varepsilon; \Delta_\varepsilon; r; \{ s \mid x \in e \} \vdash \mathtt{load}_\Delta\ F\ (a_{err}, vs)\ \delta_F\ (\delta_{a_{err}} \otimes \delta_{vs}) \Rightarrow \Theta'; ((a'_{err}, vs'), \Delta_{a_{err}})}$$

$$\frac{t \in \mathtt{dom}(vs) \quad \Theta; \varepsilon[x \mapsto t]; \Delta_\varepsilon[x \mapsto \emptyset]; r; s \vdash \mathtt{load}_\Delta\ F\ vs(t)\ \delta_F\ \delta_{vs}(t) \Rightarrow \Theta'; (v', \Delta_v)}{\Theta; \varepsilon; \Delta_\varepsilon; r; s \vdash_x \mathtt{load}_\Delta\ F\ (t, vs)\ \delta_F\ \delta_{vs} \Rightarrow \Theta'; (v', \Delta_v)}$$

$$\frac{t \notin \mathtt{dom}(vs) \quad \Theta; \varepsilon; r; s \vdash \mathtt{load}\ (\delta_F\ F) \Rightarrow \Theta'; v'}{\Theta; \varepsilon; \Delta_\varepsilon; r; s \vdash_x \mathtt{load}_\Delta\ F\ (t, vs)\ \delta_F\ \delta_{vs} \Rightarrow \Theta'; (v', \Delta)}$$

$\boxed{\Theta; \varepsilon; \Delta_\varepsilon; r; s \vdash \mathtt{store}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta'; (F', \phi')}$ "Under heap Θ, environment ε and delta environment Δ_ε, store the representation v for the specification s on filesystem F at path r, given filesystem changes δ_F and representation changes δ_v, and yield an updated filesystem F' and a filesystem validation function ϕ'."

$$\frac{\begin{array}{c}\Delta_\varepsilon|_{fv(s)} = \emptyset \quad \delta_F \searrow_F r = \emptyset \\ \Theta; \varepsilon; r; s \vdash \mathtt{sense}\ v \Rightarrow rs \quad \phi = \lambda F'.\ F \underset{rs}{=} F'\end{array}}{\Theta; \varepsilon; \Delta_\varepsilon; r; s \vdash \mathtt{store}_\Delta\ F\ v\ \delta_F\ \emptyset \Rightarrow \Theta; (F, \phi)}$$

$$\frac{\Theta; \varepsilon; r; s \vdash \mathtt{store}\ (\delta_F\ F)\ v \Rightarrow \Theta'; (F', \phi')}{\Theta; \varepsilon; \Delta_\varepsilon; r; s \vdash \mathtt{store}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta'; (F', \phi')}$$

$\boxed{s = \mathsf{M}\, s_1}$

$$\frac{\begin{array}{c}\Theta(a) = e \quad \Theta; e \Rightarrow \Theta'; (a_{err}, v) \\ \Theta'; \varepsilon; \Delta_\varepsilon; r; s \vdash \mathtt{store}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta''; (F', \phi')\end{array}}{\Theta; \varepsilon; \Delta_\varepsilon; r; \mathsf{M}\, s \vdash \mathtt{store}_\Delta\ F\ a\ \delta_F\ (\mathsf{M}_{\delta_a}\ (\delta_{a_{err}} \otimes \delta_v)) \Rightarrow \Theta''; (F', \phi')}$$

$\boxed{s = e :: s_1}$

$$\frac{\begin{array}{c}\Delta_\varepsilon|_{fv(s)} = \emptyset \quad \Theta; [\![r\ /\ e]\!]^{\varepsilon}_{Path} \Rightarrow \Theta'; r' \\ \Theta'; \varepsilon; \Delta_\varepsilon; r'; e :: s \vdash \mathtt{store}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta''; (F', \phi')\end{array}}{\Theta; \varepsilon; \Delta_\varepsilon; r; e :: s \vdash \mathtt{store}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta''; (F', \phi')}$$

$\boxed{s = \langle x : s_1, s_2 \rangle}$

$$\frac{\begin{array}{c}\Theta; \varepsilon; \Delta_\varepsilon; r; s_1 \vdash \mathtt{store}_\Delta\ F\ v_1\ \delta_F\ \delta_{v_1} \Rightarrow \Theta_1; (F_1', \phi_1') \\ \Theta_1; \varepsilon[x \mapsto v_1]; \Delta_\varepsilon[x \mapsto \delta_{v_1}]; r; s_2 \vdash \mathtt{store}_\Delta\ F\ v_2\ \delta_F\ \delta_{v_2} \Rightarrow \Theta_2; (F_2', \phi_2') \\ \phi = \lambda F'.\ \phi_1'(F_1') \wedge \phi_2'(F_2')\end{array}}{\Theta; \varepsilon; \Delta_\varepsilon; r; \langle x : s_1, s_2 \rangle \vdash \mathtt{store}_\Delta\ F\ (a_{err}, (v_1, v_2))\ \delta_F\ (\delta_{a_{err}} \otimes (\delta_{v_1} \otimes \delta_{v_2})) \Rightarrow \Theta_2; ((F_1 +\!\!+ F_2), \phi)}$$

$\boxed{s = \mathsf{P}\ e}$

$$\frac{\phi = \lambda F'.\ \mathtt{return}\ \mathit{True}}{\Theta; \varepsilon; \Delta_\varepsilon; r; \mathsf{P}\ e \vdash \mathtt{store}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta; (F, \phi)}$$

$\boxed{s = s_1?}$

$$\frac{r \notin \mathtt{dom}(\delta_F\ F) \quad \phi = \lambda F'.\ r \notin \mathtt{dom}(F')}{\Theta; \varepsilon; \Delta_\varepsilon; r; s? \vdash \mathtt{store}_\Delta\ F\ (a_{err}, \mathit{Nothing})\ \delta_F\ (\delta_{a_{err}} \otimes \emptyset) \Rightarrow \Theta; (F, \phi)}$$

$$\frac{\begin{array}{c}r \in \mathtt{dom}(\delta_F\ F) \quad \Theta; \varepsilon; \Delta_\varepsilon; r; s \vdash \mathtt{store}_\Delta\ F\ v\ \delta_F\ \delta_v \Rightarrow \Theta'; (F_1, \phi_1) \\ \phi = \lambda F'.\ \phi_1(F') \wedge e \in \mathtt{dom}(F')\end{array}}{\Theta; \varepsilon; \Delta_\varepsilon; r; s? \vdash \mathtt{store}_\Delta\ F\ (a_{err}, \mathit{Just}\ v)\ \delta_F\ (\delta_{a_{err}} \otimes \delta_v?) \Rightarrow \Theta'; (F_1, \phi)}$$

$\boxed{s = \{ s \mid x \in e \}}$

$$\frac{\begin{array}{c}\Theta; [\![e]\!]^{\varepsilon}_{\{\tau\}} \Rightarrow \Theta'; ts \quad vs = \{ t_1 \mapsto v_1, ..., t_k \mapsto v_k \} \\ \phi = \lambda F'.\ ts = \{ t_1, ..., t_k \} \wedge \bigwedge\ \phi_i(F') \\ \Theta_1; \forall t_i \in \mathtt{dom}(vs).\ \mathtt{do}\ \{ (F_i, \phi_i) \leftarrow \varepsilon[x \mapsto t_i]; \Delta_\varepsilon[x \mapsto \emptyset]; r; s \vdash \mathtt{store}_\Delta\ F\ vs(t_i)\ \delta_F\ \delta_{vs}(t_i); \mathtt{return}\ (F_1 +\!\!+ \ldots +\!\!+ F_k, \phi) \} \Rightarrow \Theta_2; (F', \phi')\end{array}}{\Theta; \varepsilon; \Delta_\varepsilon; r; \{ s \mid x \in e \} \vdash \mathtt{store}_\Delta\ F\ (a_{err}, vs)\ \delta_F\ (\delta_{a_{err}} \otimes \delta_{vs}) \Rightarrow \Theta_2; (F', \phi')}$$

$\boxed{\Theta; \varepsilon; r; s \vdash \mathtt{sense}\ v \Rightarrow rs}$ "Sensitivity of a forest specification in respect to a representation"

$$\frac{\Theta(a) = e \quad \Theta; e \Rightarrow \Theta'; v \quad \Theta'; \varepsilon; r; s \vdash \mathtt{sense}\ v \Rightarrow rs}{\Theta; \varepsilon; r; \mathsf{M}\ s \vdash \mathtt{sense}\ a \Rightarrow rs}$$

$$\frac{\Theta; \varepsilon; r; s \vdash \mathtt{sense}\ v \Rightarrow rs}{\Theta; \varepsilon; r; e :: s \vdash \mathtt{sense}\ v \Rightarrow \{ r \} \cup rs}$$

$$\frac{\Theta; \varepsilon; r; s_1 \vdash \mathtt{sense}\ v_1 \Rightarrow rs_1 \quad \Theta; \varepsilon[x \mapsto v_1]; r; s_2 \vdash \mathtt{sense}\ v_2 \Rightarrow rs_2}{\Theta; \varepsilon; r; \langle x : s_1, s_2 \rangle \vdash \mathtt{sense}\ (a_{err}, (v_1, v_2)) \Rightarrow rs_1 \cup rs_2}$$

$$\frac{}{\Theta; \varepsilon; r; \mathsf{P}\ e \vdash \mathtt{sense}\ v \Rightarrow \{ \}}$$

$$\frac{}{\Theta; \varepsilon; r; s? \vdash \texttt{sense}\ (a_{err}, \mathit{Nothing}) \Rightarrow \{\, r \,\}}$$

$$\frac{\Theta; \varepsilon; r; s \vdash \texttt{sense}\ v \Rightarrow rs}{\Theta; \varepsilon; r; s? \vdash \texttt{sense}\ (a_{err}, \mathit{Just}\ v) \Rightarrow \{\, r \,\} \cup rs}$$

$$\frac{vs = \{\, t_1 \mapsto v_1, ..., t_k \mapsto v_k \,\} \quad \forall\, 1 \in \{i, ..., k\}.\ \Theta; \varepsilon[x \mapsto t_i]; r; s \vdash \texttt{sense}\ v_i \Rightarrow r_i}{\Theta; \varepsilon; r; \{s \mid x \in e\} \vdash \texttt{sense}\ (a_{err}, vs) \Rightarrow \bigcup\ r_i}$$

Theorem B.1 (Incremental Load Soundness)**.** *If*

$$\Theta; \varepsilon; r; s \vdash \texttt{load}\ F_1 \Rightarrow \Theta_1; v_1$$
$$\Theta_1; v_1 \xrightarrow{\delta_{v_1}} \Theta_2; v_1'$$
$$\Theta_2; \varepsilon'; \Delta_\varepsilon; r; s \vdash \texttt{load}_\Delta\ F_1\ v_1'\ \delta_{F_1}\ \delta_{v_1} \Rightarrow \Theta_3; (v_2, \Delta_{v_1'})$$
$$\Theta_1; \varepsilon'; r; s \vdash \texttt{load}\ (\delta_{F_1}\ F_1) \Rightarrow \Theta_4; v_3$$

then $v_2\ {}_{\Theta_3}\!\overset{err}{\sim}_{\Theta_4}\ v_3$ *and* $valid(v_2)\ {}_{\Theta_3}\!\overset{err}{\sim}_{\Theta_4}\ valid(v_3)$.

$$\begin{tikzcd}
F_1 \arrow[r, "\texttt{load}"] \arrow[d, squiggly, "\delta_{F_1}"'] \arrow[ddr, Rightarrow, "\texttt{load}_\Delta"] & v_1 \arrow[d, squiggly, "\delta_{v_1}"] \\
F_2 \arrow[d, "id"'] & v_1' \arrow[d, squiggly, "\Delta_{v_1'}"] \\
F_2 \arrow[r, "\texttt{load}"] & v_2
\end{tikzcd}$$

Lemma 1 (Incremental Load Stability)**.** $\Theta; \varepsilon; \Delta_\varepsilon; r; \mathsf{M}\ s \vdash \texttt{load}_\Delta\ F\ a\ \delta_F\ (\mathsf{M}_\emptyset\ \delta_v) \Rightarrow \Theta'; (a, \Delta_a)$

Theorem B.2 (Incremental Store Soundness)**.** *If*

$$\Theta; \varepsilon; r; s \vdash \texttt{store}\ F\ v_1 \Rightarrow \Theta_1; (F_1, \phi_1)$$
$$\Theta_1; v_1 \xrightarrow{\delta_{v_1}} \Theta_2; v_2$$
$$\Theta_2; \varepsilon'; \Delta_\varepsilon; r; s \vdash \texttt{store}_\Delta\ F_1\ v_2\ \delta_{F_1}\ \delta_{v_1} \Rightarrow \Theta_3; (F_2, \phi_2)$$
$$\Theta_2; \varepsilon'; r; s \vdash \texttt{store}\ (\delta_{F_1}\ F_1)\ v_2 \Rightarrow \Theta_4; (F_3, \phi_3)$$

then $F_2 = F_3$ *and* $\phi_2(F_2) = \phi_3(F_3)$.

$$\begin{tikzcd}
F \arrow[d, squiggly] & v_1 \arrow[d, "id"] \arrow[dl, "\texttt{store}"] \\
F_1 \arrow[r, "\texttt{load}"'] \arrow[d, squiggly, "\delta_{F_1}"'] & v_1 \arrow[d, squiggly, "\delta_{v_1}"] \arrow[ddl, Rightarrow, "\texttt{store}_\Delta"'] \\
F_1' \arrow[d, squiggly] & v_2 \arrow[d, "id"] \arrow[dl, "\texttt{store}"] \\
F_2 \arrow[r, "\texttt{load}"'] & v_2
\end{tikzcd}$$