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

excerpt from [3]: Although database systems are equipped with more advanced and secure data management features such as transactional atomicity, consistency, durability, manageability, and availability, lack of high performance and throughput scalability for storage of unstructured objects, and absence of standard filesystem-based application program interfaces have been cited as primary reasons for content management providers to often prefer existing filesystems or devise filesystem-like solutions for unstructured objects.

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. offers a load/store interface, placing full responsibility for explicitly managing the filesystem with the application.
- 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)

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

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

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?)

Specific use cases: LHC

Network logs

Dan's scientific data

what filesystem API gives me true transactionality? this is, TxForest is currently a library, but we would need to be able to lock concurrent modifications to shared resources at commit time; to compute all symlinks under a path. advantages of TxForest over DBs: the user gets to declare the schema according to which he wants to read the data; he chooses the abstraction, instead of a fixed tuple store.

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

```
data Balance = ...
type Balance_md = ...

data Accounts
instance TxForest () Accounts (FileInfo, [(FilePath, Account)]) where ...

data Account
instance TxForest () Account ((FileInfo, Balance_md), Balance) where ...
```

```
[forest |
  type Universal_d = Directory
    { ascii_files  is [f :: TextFile      | f ← matches (GL "*"), (kind f_att == AsciiK)]
    , binary_files is [b :: BinaryFile    | b ← matches (GL "*"), (kind b_att == BinaryK)]
    , directories  is [d :: Universal_d   | d ← matches (GL "*"), (kind d_att == DirectoryK)]
    , symlinks     is [s :: Link          | s ← matches (GL "*"), (is_just (sym_Link s_att))]
    }
|]
```

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 filesystem 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]

$$atomic :: FTM\ a \to IO\ a$$

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:

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

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, filesystem 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:

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

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 **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 $ftm2 :: a \to FTM\ b$. 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:

$$orElse :: FTM\ a \to FTM\ a \to FTM\ a$$

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 filesystem.

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:

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

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

```
tryRead = do
  { exists ← ...find file ...
  ; if (not exists) then throw FileNotFound else return ()
  ; ...read file... }
```

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

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.:

```
catch tryRead
  (λFileNotFound → return ... default...) tryRead
```

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 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 filesystem. 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:

```
class TxForest args ty rep | ty → rep, ty → args where
  new         :: args → FilePath → FTM fs ty
  read        :: ty → FTM rep
  writeOrElse :: ty → rep → b
                 → (Manifest → FTM fs b) → FTM fs b
```

In this signature, *ty* is an opaque transactional variable type that uniquely identifies a user-declared Forest type. Each transactional variables provides a window into the filesystem, shaped as a plain Haskell representation type *rep*. 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*.

[Shall we instead have type/data declaractions as in Pads, where only data declarations create new tx variables, to provide users more control? E.g., we currently have to create variables for top-level directory records since these need to be declared as top-level types.]

Creation The transactional forest programming style makes no distinction between data on the filesystem 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 filesystem. This operation does not have any effect on the filesystem 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 **??**:

```
do
  accs :: Accounts ← new () "/var/db/accounts"
  (accs_info, accs_rep) ← read accs
  let acc1 :: Account = fromJust (lookup "account1" accs_rep)
  ((acc1_info, acc1_md), Balance balance) ← read acc1
  return balance
```

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:

```
[forest |
  type Accounts = [a :: File Balance | a ← matches (GL "*")]
 |]
```

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 filesystem 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 filesystem and a report of the inconsistencies. We can easily define more convenient derived forms of *writeOrElse*:

```
  -- optional write
tryWrite :: TxForest args ty rep ⇒ ty → rep → FTM ()
tryWrite t v = writeOrElse t v () (const (return ()))
  -- write or restart the transaction
writeOrRetry :: TxForest args ty rep ⇒ ty → rep → () → FTM ()
writeOrRetry t v = writeOrElse t v () (const retry)
  -- write or yield an error
writeOrThrow :: (TxForest args ty rep, Exception e) ⇒ ty → rep → ()
writeOrThrow t v e = writeOrElse t v () (const (throw e))
```

A typical example of an inconsistent representation is when a Forest description refers to the same file more than once, to describe it in multiple ways, and the user attempts to write conflicting data in each occurrence. For instance, in Forest we may describe a symbolic link to an ASCII file both as a *SymLink* and a *Text* file:

```
type Folder = Directory {
  {link  is "README" :: SymLink
  , notes is "README" :: File Text
  , ...}
```

Here, the file information for the *link* and *notes* fields must match.

Akin to reactive environments like (bidirectional) spreadsheets [**?**], each write takes *immediate* effect on the (transactional snapshot of the) filesystem: an update on a variable is automatically propagated to the filesystem, eventually triggering the update of other variables dependent on common parts of the filesystem. We can observe this data flow by defining two accounts pointing to the same file and writing to one of them:

```
acc1 :: Account ← new () "/var/db/accounts/account"
acc2 :: Account ← new () "/var/db/accounts/account"
(acc_md, Balance balance) ← read acc2
tryWrite acc1 (acc_md, Balance (balance + 1))
(acc_md', Balance balance') ← read acc2
```

By incrementing the balance of *acc1*, we are implicitly incrementing the balance of *acc2* (if the write succeeds, then $balance' = balance + 1$). Note that even if we attempt to write different balances to each variable, in sequence:

```
tryWrite acc1 (acc_md, Balance 10)
tryWrite acc2 (acc_md, Balance 20)
```

it is always the case that *acc1* and *acc2* have the same balance, and there is no inconsistency since the first write propagates to both variables before the second write occurs.

4.3 Validation

As Forest lays a structured view on top of a semi-structured filesystem, 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 ⇒ ty → FTM ForestErr
```

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

```
data ForestErr = ForestErr
  { numErrors :: Int
  , errorMsg  :: Maybe ErrMsg }
```

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

```
validRead :: TxForest args ty rep ⇒ ty → FTM rep
validRead ty = do
  rep ← read ty
  err ← validate ty
  if numErrors err ≡ 0
    then return rep
    else throw (ForestError err)
```

4.4 Standard filesystem 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:

```
do
  badAcc :: Account ← new () "/var/db/accounts/account"
  (acc_info, Balance balance) ← read badAcc
```

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:

```
newAccount path balance = do
  newAcc :: Account ← new () path
  tryWrite newAcc (validFileInfo path, Balance balance)
```

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

```
delAccount acc = do
  tryWrite acc (invalidFile, Balance 0)
```

The takeaway lesson is that the *FileInfo* metadata actually determines whether a directory, file or symbolic link exists or not in the filesystem, 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:

```
rm :: TxForest args ty rep ⇒ ty → FTM ()
```

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

```
copyAccount srcpath tgtpath = do
  src :: Account ← new () srcpath
  tgt :: Account ← new () tgtpath
  (info, balance) ← read src
  tryWrite tgt (info {fullpath = tgtpath}, balance)
```

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:

```
cpOrElse :: TxForest args ty rep ⇒ ty → ty → b
         → (Manifest → rep → FTM fs b) → FTM fs b
```

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:

```
[forest |
  type NamedAccounts (acc :: String) =
    [a :: Account | a ← matches (GL (acc ++ "*"))]
 |]
```

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:

```
do
  src :: Accounts ← new "account" "/var/db/accounts"
```

$tgt :: Accounts \leftarrow new$ `"acc" "/var/db/accs"`
$tryCopy\ src\ tgt$

4.5 Read-only transactions

explain the need for monadic quotation and give an example

however, this introduces problems therefore embedded expressions are read-only, to prevent side-effects during loads/stores.

transactional operations ($retry$,$orElse$,$atomic$) are done in write mode ($FTM\ RW\ a$).

$read$ and new in any mode ($FTM\ mode\ a$).

$FTM\ RO\ a$ can only read data from the fs

$FTM\ RW\ a$ can write data to the fs

expressions statically restricted to just perform a series of new/read operations.

for implementation purposes, it also allows the transaction manager to distinguish read-only transactions

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. We split our presentation into three possible designs, with increasing levels of incremental support and complexity.

5.1 Transactional Forest

Original STM interface We have implemented TxForest as a domain-specific variant of `STM` Haskell [2], and inherit the same transactional mechanism based on *optimistic concurrency control*: each transaction runs in a (possibly) different thread and keeps a private log of reads and writes (including the tentatively-written data) to *shared resources*, and reads within a transaction first consult its log so that they see preceding writes. Once finished, each transaction validates its log against previous transactions that committed before its starting time and, only if no write-read conflicts are detected, commits its writes permanently; otherwise, it is re-executed. These validate-and-commit operations are guaranteed to run $atomic$ally in respect to all other threads by relying on per-shared-resource locks (no locks are used during the transaction's execution): the transaction waits on the sorted sequence of read resources to be free (to ensure that it sees the commits of concurrently writing transactions) and acquires the sorted sequence of written resources. They are *disjoint-access parallel* (meaning that transactions with non-overlapping writes run in parallel) and *read parallel* (meaning that transactions that only read from the same resources run in parallel). These wait-and-acquire sequences are repeatedly attempted atomically, without interruption by the Haskell scheduler, and implemented over GHC's lightweight concurrency substrate [**?**].

Blocking transactions ($retry$) validate their log and register themselves in wait-queues attached to each read resource; updating transactions unblock any pending waiters. Nested transactions ($orElse$) work similarly to normal transactions: writes are recorded only to a nested log and reads consult the logs of nested and all enclosing transactions. Validating a nested transaction also implies validating all enclosing transactions. If the first alternative retries, then the second alternative is attempted; if both retry, then both logs are validated and the thread will wait on the union of the read resources. Exceptional transactions ($throw$) must also validate the log before raising an exception to the outside world; on success, they rollback all modifications except for newly-created transactional variables; on failure, they retry. A more detailed account, including a complete formal semantics, is given in [2].

Transaction logs The main difference from `STM` Haskell to TxForest is that the shared resources are not mutable memory cells in the traditional sense, but paths in the filesystem. [2] This is to say that, although users manipulate structured filestores, all the in-memory data structures are local to each transaction, and only filesystem operations need to be logged for commit.

The concurrent handling of file paths, however, is subtle in the presence of symbolic links –the identity of a path is not unique (as different paths may refer to the same real path) nor stable (since the real path depends on the current symbolic link configuration)– making it harder to identify conflicts between transactions and to properly lock resources. For example, one transaction may read a file whose path is concurrently modified by other transaction. Therefore, our transaction logs keep special track of symbolic link modifications and we perform all file operations over "canonical" file paths, calculated against the transaction log while marking each resolved link as read.

Round-tripping functions In TxForest, each transactional variable is an in-memory data structure that reflects the content of a particular filestore, declared from the respective Forest description type with given arguments and root path. Behind the scenes, the transactional engine is responsible for preserving the abstraction, and keeping each variable "in sync" with the latest transactional snapshot of the filesystem, such that changes on the filesystem are propagated to the affected in-memory filestore variables, and writes to variables move the filesystem snapshot forward.

This task is performed by a coupled pair of `load` and `store` functions. Their precise definitions and formal semantics is given in Appendix A. Informally, each Forest variable is implemented as a "thunk" that lazily computes visible data (denoting the content and metadata of the filestore) and hidden data (remembering errors during validation).

The `load` function for a top-level variable generates a top-level thunk that, once evaluated, recursively reads data from the (transactional snapshot of the) filesystem, building a thunk for each level of structure. Error information is computed behind an additional thunk to guarantee that validation is only performed when explicitly demanded by the user.

The `store` function strictly traverses a nested structure of in-memory thunks and updates the (transactional snapshot of the) filesystem to reflect the same content, returning an additional *validator* that can test the updated filesystem for inconsistencies during storing.

These two functions are carefully designed so that they preserve data on round trips: loading a filestore and immediately storing it back always succeeds and keeps the filesystem unchanged; and storing succeeds as long as loading the updated filesystem yields the same in-memory representation.

Transactional variables In TxForest, each transaction keeps a local filesystem snapshot with a unique thread-local version number and a log of tentative updates over the real filesystem.

When users create a new transactional variable, the `load` function is called to create the corresponding thunk with a suspendend computation that always loads data from the latest version of the filesystem. Note that, due to laziness, no data is actually loaded. These thunks, acting as transactional variables, can be concurrently accessed by multiple transactions. Each transacation keeps a memoization table mapping variables to the latest read values at a particular filesystem version. We implement them as weak hash tables, allowing the Haskell garbage collector to purge older entries.

When a transaction $read$s a transactional variable it first checks the memoization table for a value for the current filesystem, other-

[2] STM maintains a log with the old value held in a memory cell and the new value written to it by the transaction, and validation test if they are pointer-equal. We do not remember old content of file paths, nor test for equality.

wise the associated level of data is loaded from the current filesystem snapshot, and adds a new memo entry with the read value at the current version.

A call to $writeOrElse$ starts by making a copy of the current filesystem snapshot and adding an entry to the memoization table of the respective variable mapping the next filesystem version to the newly written value. It then invokes the `store` function (remembering the creation-time arguments and root path) to update the filesystem log under the new version and runs the resulting validator; on inconsistencies, the transaction rolls back to the backed up filesystem snapshot and executes the user-supplied alternative action instead.

5.2 Incremental Transactional Forest

We have described all the components for a complete implementation of TxForest, but not a very efficient one. To understand its limitations, imagine that a transaction maintains two completely unrelated variables and reads the first, writes new content to the second, and reads the first again. Since the two reads occur between a filesystem change, our simple memoization mechanism will fail, and the same content will be redundantly loaded from the filesystem twice. Even worse, writes in TxForest force a deep evaluation of the in-memory filestore, compromising the convenient laziness properties of the runtime system. For example, if a transaction reads a directory variable and immediately writes the read value to the same variable, the underlying `store` function will strictly traverse the filestore to redundantly overwrite sub- files and directories, even though their content hasn't actually changed.

Round-tripping functions The problem in both examples is that the `load` and `store` functions used by the runtime system are agnostic to modifications made to the filesystem or to the in-memory data, and execute from scratch every time with a running "footprint" proportional to the size of the Forest description. Being Forest an embedded DSL in Haskell, we can exploit domain-specific knowledge to design incremental round-tripping functions, intuitively named $\texttt{load}_\Delta$ and $\texttt{store}_\Delta$, with "footprint" proportional to the size of the actual updates. Especially since transactions are already equipped with the machinery to keep logs of modifications, we can extend our runtime system to make use of the incremental functions in place of their non-incremental counterparts. In this section we informally discuss the necessary extensions. The formal ingredients for an incremental Forest load/store semantics are developed in Appendix B.

File system updates In order to support incrementality within individual transactions, each transaction shall be able to identify the updates that occurred since a previous point in time (denoted by a filesystem version). We thus split the sequences of writes recorded in transaction logs according to their increasing filesystem versions.

The key behind incrementality is to exploit the locality of updates, so that the algorithm can distinguish affected structures that need to be updated from unaffected ones that are already up-to-date. In the context of filesystems, with updates as sequences of operations on file paths, locality checking boils down to path inclusion: seeing the filesystem as a tree, an update on path r' is local to another path r if r' is a subpath of r.

However, as filesystems are in fact graphs, a branch in the filesystem may be affected by changes made to distinct branches, by following symbolic links. This requires our transaction manager to keep a materialized record of all the symbolic links in the filesystem. Although finding all symbolic links in traditional filesystems is an expensive task that involves traversing the whole file hierarchy, it can be amortized by running once when the transaction manager is initialized, and having transactions updating the necessary symbolic link records. This cost becomes negligible in a database-centric model, in which an always-on server assumes full control of the filesystem and accepts transactions issued by clients.

Filestore updates Additionally to filesystem updates, transactions must also recognize the data modifications that have been performed on a given in-memory data structure since a particular filesystem version. For individual variables, our transaction logs already keep entries with the latest value read from or wrote to them, annotated with the corresponding filesystem version. But filestores often have multiple levels of nested variables (e.g., an accounts directory contains multiple account variables) and data edits may occur deep in the structure (e.g., the user updates one specific account).

In order to support incremental update propagation, a variable must also be aware of modifications to its underlying child variables. We do so by maintaining an extra acyclic parenthood graph between thunks and an extra field with the version of the last modification made to (the underlying structure of) each variable. A new variable is created with no parent nor child edges, and set as last modified at the current filesystem version. When a variable is read or written, we remove its child edges from the graph, add new parent edges for its immediate child variables, and recursively set the modification version for all its parent variables (itself included) to the current filesystem version.

Transactional variables In the incremental variant of TxForest, new variables are created as before, with a call to `load`.

When $read$ing a variable, we first search for a memoized value; if none is found, we load the contents from the filesystem as before. If an old value is found, we compute the updates since the previous filesystem version, by filtering the logged filesystem updates in the corresponding interval that affect the variable's root path and by retrieving the variable's latest modification version. We then invoke $\texttt{load}_\Delta$ (assuming that top-level arguments may have changed) to incrementally repair the memoized in-memory representation, and finish by reading the content of the variable as before. The $\texttt{load}_\Delta$ function strictly traverses the filesystem and the in-memory data according to the Forest description: if there are no filesystem nor data updates (i.e., the latest modified version of the top-level variable matches the version of the memoized value), it stops; it also stops on variables with no memoized data, since there is no old value to be repaired; otherwise, it repairs the memoized value according to the updates and proceeds recursively.

When writing to a variable ($writeOrElse$), we first search for a memoized value; if none is found, we write the new contents as before. If an old value is found, we compute the updates since the memoized value and invoke $\texttt{load}_\Delta$ to make sure that all memoized data for the filestore is up-to-date with the current filesystem version. We then backup the transaction log, increment the filesystem version, write the new content to the variable and invoke $\texttt{store}_\Delta$ (with empty filesystem updates and only the user's write to the top-level variable as a data update) to incrementally repair the filesystem. Likewise, the $\texttt{store}_\Delta$ function strictly traverses the filesystem and the in-memory data according to the Forest description: if there are no updates, it stops; otherwise, it modifies the filesystem to conform to the new content and proceeds recursively. Note that, in the general case, when $\texttt{store}_\Delta$ stops the generated validator still needs to verify that the unmodified fragment of the filesystem is consistent with other $\texttt{store}_\Delta$ traversals that may have modified overlapping fragments of the filesystem —in order to guarantee that arbitrary data dependencies in a Forest specification are correctly enforced. We can skip these checks in our particular application, as long as we only call $\texttt{store}_\Delta$ with a top-level data modification.

5.3 Log-structured Transactional Forest

Thus far, our implementation of TxForest only supports incremental updates locally to each transaction. This has the potential to

greatly speed-up larger transactions that combine several filestore operations, but is ineffective for smaller transactions. For example, it offers no incrementality if each single filestore operation is performed in a separate transaction. It is also non-optimal for more common scenarios that involve multiple concurrent transactions trying to access a shared resource, e.g., a set of configuration files: each transaction will have to load all necessary files into memory, perform the respective modifications, and discard all the in-memory data at the end; such loading work is often redundant, even more for workloads dominated by read-only transactions where the filesystem seldom changes between consecutive transactions.

Unfortunately, this is how far TxForest can reasonably go on top of a conventional filesystem. As in any OCC method, each transaction builds an interim timeline of filesystem updates that is preserved for the duration of concurrently running transactions. Consequently, there is no global timeline that records the history of filesystem modifications to which in-memory data can be chronologically affixed across transactions.

Log-structured filesystem This is precisely one of the main advantages of a log-structured filesystem, in which updates do not overwrite existing data but rather issue continuous snapshots, that can be made accessible as a log that records the history of modifications on the filesystem since an initial point in time. To provide inter-transaction incrementality, we have explored a variant of TxForest implemented on top of NILFS, a log-structured filesystem for Linux supporting read-only snapshots.

Transactional logs The main difference in our third design is that transactions are logically executed over particular snapshots of the filesystem. When initialized, each transaction acquires the version number of the latest snapshot from the NILFS filesystem. Reads on the filesystem are instead performed against the initial snapshot; and groups of writes create advancing transaction-local snapshots over the initial physical snapshot. On success, the transaction commits its modifications over the current filesystem (that may be at newer snapshot than the transaction's snapshot due to concurrent updates) and requests the creation of a new NILFS snapshot combining the concurrent and the transaction-local updates.

Transactional variables We also refine variables to store memoized content that was previously read at an older NILFS snapshot. This enables a transaction to incrementally reuse filestores loaded by previous transactions. The filesystem delta is computed as the difference between the memoized and the transaction's NILFS snapshots, prepended to the transaction-local tentative updates. Once finished, the transaction commits its local data updates to the global transactional variables; to ensure that all the committed data is bound to externally visible NILFS snapshots, we first use $\texttt{load}_{\Delta}$ to synchronize all the transaction-local memoized data with the latest filesystem snapshot —the one for which we will later request a new NILFS snapshot. Since multiple transactions may now concurrently read and write the content of the same transactional variables, each $atomic$ block must acquire extra per-variable locks during the validate-and-commit phase.

Read-only transactions A side-benefit of using a snapshotting filesystem is that every transaction starts from a possibly outdated, but nevertheless consistent, snapshot of the filesystem. Following this observation, read-only transactions can be optimized to bypass all the logging, validation and committing mechanisms, as if they happened to run before all other concurrent transactions [**?**]. Read-write transactions need to anyway validate their read file paths against concurrent writes, to guarantee that they are assigned a position in the serial order.

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`.

[3] K. Kunchithapadam, W. Zhang, A. Ganesh, and N. Mukherjee. Dbfs and securefiles. 2011.

[] N. Macedo, H. Pacheco, N. Rocha Sousa, and A. Cunha. Bidirectional spreadsheet formulas. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, pages 161–168, July 2014.

A. Forest Semantics

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

$$\frac{}{r \in \cdot} \quad \frac{}{r \in r} \quad \frac{r \in r'}{r \mathbin{/} u \in r'}$$

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

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

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

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

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

$\mathcal{R}[\![\cdot]\!]$ 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

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

$v_1 \ {}_{\Theta_1}{\sim}_{\Theta_2}\ v_2$ 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."

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

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

$\boxed{s = k}$

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

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

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

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

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

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

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

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

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

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

$\boxed{s = s_1?}$

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

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

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

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

$\boxed{\Theta;\varepsilon;r;s \vdash \texttt{store}\ F\ v \Rightarrow \Theta';(F',\phi')}$ "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'."

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

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

$\boxed{s = k}$

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

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

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

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

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

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

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

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

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

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

$\boxed{s = s_1?}$

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

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

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

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

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*

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

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

$\boxed{s = s_1?}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\boxed{s = s_1?}$

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

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

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

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

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

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

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

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

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

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

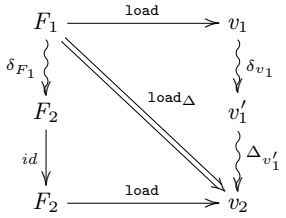
$$\frac{\Theta; \varepsilon; r; s \vdash \texttt{sense}\ v \Rightarrow rs}{\Theta; \varepsilon; r; s? \vdash \texttt{sense}\ (a_{err}, \mathit{Just}\ v) \Rightarrow \{\, r \,\} \cup rs}$$

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

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



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