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.

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

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 $< |e| >$

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, and can be understood independently from the concrete programming artifacts that can be 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 to concurrently manipulate structured filestores.

4.1 Composable transactions

As an embedded domain-specific language in Haskell, the inspiration for TxForest is the widely popular *software transactional memory* (`STM`) library, that provides a small set of highly composable operations to define the key facilities 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 an $atomic$ 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, the Haskell prelude functions:

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

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

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

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 shall 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, being FTM and IO different types, the Haskell type system effectively prevents non-transactional actions to be run inside a transaction. This is a valuable guarantee, and one that is not commonly found in transactional libraries for mainstream programming languages without a very expressive type system.

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

$$\mathit{retry} :: FTM\ a$$

Conceptually, *retry* cancels the current transaction, without emitting any errors, and schedules it to be retried at a later time. An efficient implementation waits for some of the shared filestore fragments read by the transaction to be updated before retrying.

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

$$\begin{array}{l} \mathit{wait} :: FTM\ Bool \rightarrow FTM\ a \rightarrow FTM\ a \\ \mathit{wait}\ b\ c\ a = \mathbf{do}\ \{ b \leftarrow p; \mathbf{if}\ b\ \mathbf{then}\ \mathit{retry}\ \mathbf{else}\ a \} \end{array}$$

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

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

to run a transaction $\mathit{ftm1} : FTM\ a$ and pass its result to a transaction $\mathit{ftm2} :: a \rightarrow FTM\ b$. 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:

$$\mathit{orElse} :: FTM\ a \rightarrow FTM\ a \rightarrow FTM\ a$$

This combinator performs a left-biased choice: if first runs transaction *ftm1*, tries *fmt2* if *ftm1* retries, and the whole action retries if *ftm2* retries. It can be useful, for example, to read either one of two files depending on the current configuration of the file system.

Note that *orElse* provides an elegant mechanism to define 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 an alternative *ftm2* in case *fmt1* retries:

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

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:

$$\begin{array}{l} \mathit{throw} :: \mathit{Exception}\ e \Rightarrow e \rightarrow FTM\ a \\ \mathit{catch} :: \mathit{Exception}\ e \Rightarrow FTM\ a \rightarrow (e \rightarrow FTM\ a) \rightarrow FTM\ a \end{array}$$

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

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

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.:

$$\mathit{catch}\ \mathit{tryRead}\ (\lambda \mathit{FileNotFound} \rightarrow \mathit{return}\ ...\ \mathbf{default} ...)\ \mathit{tryRead}$$

4.2 Transactional variables

$FTM\ a$ denotes a transactional action that returns a value of type a. Complex transactions can be defined by composing FTM actions, and run *atomically* as an IO action.

arbitrary pure code

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

this is precisely where we deviate from original STM

a forest variable is (conceptually) a path in the file system

Up until now, we have only seen how to compose individual transactions, but not how to do anythign meaningful with shared data!

The forest programming style draws no distinction between data represented on disk and in memory.

The transactional forest compiler generates several Haskell types and functions from every forest type declaration, aggregated as an instance of the *TxForest* class:

Programmers can manipulate in-memory representations as if they were working on the filestore itself.

Each user-declared forest type *ty* corresponds to a transactional variable that holds a representation of type *rep*.

$$\begin{array}{l} \mathbf{class}\ \mathit{TxForest}\ \mathit{args}\ \mathit{ty}\ \mathit{rep}\ \mathbf{where} \\ \quad \mathit{new} :: \mathit{args} \rightarrow \mathit{FilePath} \rightarrow FTM\ \mathit{fs}\ \mathit{ty} \\ \quad \mathit{read} \quad :: \mathit{ty} \rightarrow FTM\ \mathit{rep} \\ \quad \mathit{writeOrElse} :: \mathit{ty} \rightarrow \mathit{rep} \rightarrow b \rightarrow (\mathit{WriteErrors} \rightarrow FTM\ \mathit{fs}\ b) \rightarrow FTM \end{array}$$

new creates a new forest transactional variable for the specification found in the *TxForest* context, with arbitrary arguments and a root path. *read* reads the associated fragment of the filesytem into an in-memory representation data structure. Users can manipulate these structures as they would in regular Haskell programs, and eventually perform FS modifications by writing a new representation to a transactional variable. writes may fail if the provided data is not a faithful representation of the filestore for the specification under consideration.

WriteErrors have nothing to do with transactional errors and account for the inconsistencies that can arise when a programmer attempts to write an erroneous in-memory representation to the filestore. For example, attempting to write conflicting data to the same file or a text file to a specification of a directory structure.

The rep of a variable may contain other variables such as a directory containing a list of other Forest types. TODO by Hugo: write a simple programming example.

Notice that we can have multiple variables (possibly with different specs) "connected" to the same fragment of a filesystem. This can cause WriteErrors, as noted above, and the values of the two will be interdependent. However, variables only depend on each other within a transaction, not across transactions (until a transaction is committed that is).

NOTE by JD: Not sure what this fragment of a sentence meant Hugo: this can cause inter

NOTE by JD: I kind of see what you're trying to say below Hugo (explaining why we need fileinfo I guess?), but the first line doesn't really make sense.

We have a sort of mismatch: Transactional variables for type declarations VS fileinfo for directories/files. Since forest always fills in default data for non-existing paths, the fileinfo actually determines whether a directory/file exists or not in the real FS. E.g. to delete a file we need to mark its fileinfo as invalid, and to create a file we need to define clean, valid fileinfo for it.

4.3 Validation

Validation helps programmers detect inconsistencies between the data they are trying to write to the filesystem and the constraints they have specified through Forest. In order to detect these sorts of errors, which we allow them to make should they care to, we provide a validate function, returning all such errors.

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

4.4 Standard filesystem operations

not a problem of expressiveness, for convenience

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

NOTE by JD: Does rm actually remove a filepath or a specification? Or are these practically the same in this case (which would mean that if you only specify part of what's in a folder, the unspecified parts would be removed using this command)?

This command lets the programmer remove a filepath by writing invalid fileinfo and default data to it. In order to avoid a loss of information, the default data needs to be precisely the data that is generated by forest. If we are removing a directory, we need to make sure that its content is the empty list; a non-existing directory with content inside is not a valid snapshot of a FS, but a valid haskell value nonetheless. This is cumbersome to do manually for arbitrary specs that touch multiple files/directories, which is why we provide this primitive operation that generates the appropriate default data and performs the removal.

4.5 Lazy Forest I/O

$$cpOrElse :: TxForest\ args\ ty\ rep \Rightarrow ty \rightarrow ty \rightarrow b \rightarrow ([\,WriteErrors\,] \rightarrow FTM\ fs\ b) \rightarrow FTM\ fs\ b$$

This command lets the programmer copy a forest specification. While copying a single file by hand is simple (read, copy the contents, update the fileinfo, write), copying a directory is significantly more cumbersome because we have to recursively copy each child variable and update its fileinfo accordingly. Therefore, we provide this primitive operation. It may fail because the data that we are trying to write may not be consistent with the specification for the target arguments and path. For example, a specification with a boolean argument that loads file x or y, with source argument True and target argument False.

NOTE by JD: Not sure I quite understand the example of where it may fail Hugo.

5. Implementation

5.1 Transactional Forest

(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

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

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

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

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

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

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

s	$\mathcal{R}[\![s]\!]$	$\mathcal{C}[\![s]\!]$
$\mathsf{M}\,s$	$\mathsf{M}\,(\mathit{Err}\ (\mathcal{R}[\![s]\!]))$	$\mathsf{M}\,(\mathit{Err}\ (\mathcal{R}[\![s]\!]))$
$k_{\tau_1}^{\tau_2}$	$\mathit{Err}\ (\tau_2, \tau_1)$	(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."

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

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

$\boxed{s = k}$

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

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

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

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

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

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

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

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

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

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

$\boxed{s = s_1?}$

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

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

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

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

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

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

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

$\boxed{s = k}$

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

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

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

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

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

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

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

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

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

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

$\boxed{s = s_1?}$

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\boxed{s = s_1?}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\boxed{s = s_1?}$

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

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

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

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

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

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

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

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

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

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

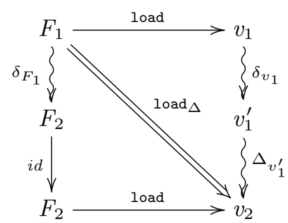
$$\frac{\Theta; \varepsilon; r; s \vdash \texttt{sense}\ v \Rightarrow rs}{\Theta; \varepsilon; r; s? \vdash \texttt{sense}\ (a_{err}, \mathit{Just}\ v) \Rightarrow \{ r \} \cup rs}$$

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

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



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