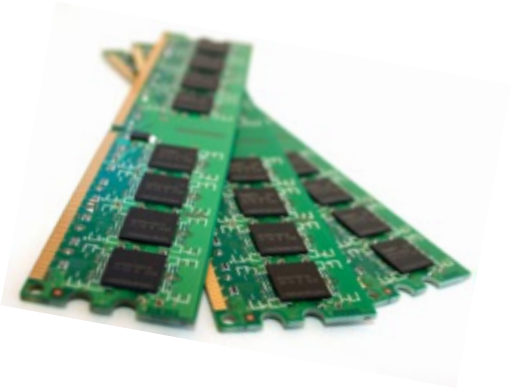
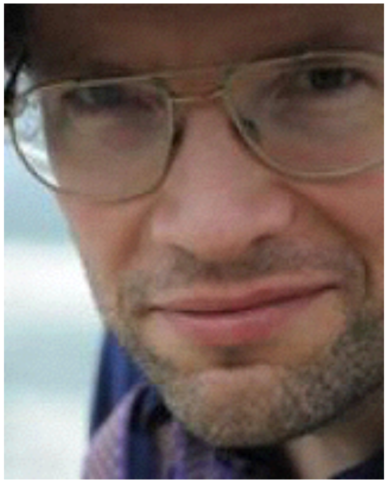
# Transactional Forest
# Strong Consistency for File Stores

Jonathan DiLorenzo (Cornell)
Kathleen Fisher (Tufts)
Nate Foster (Cornell)
Hugo Pacheco (Cornell)
Richard Zhang (Cornell)

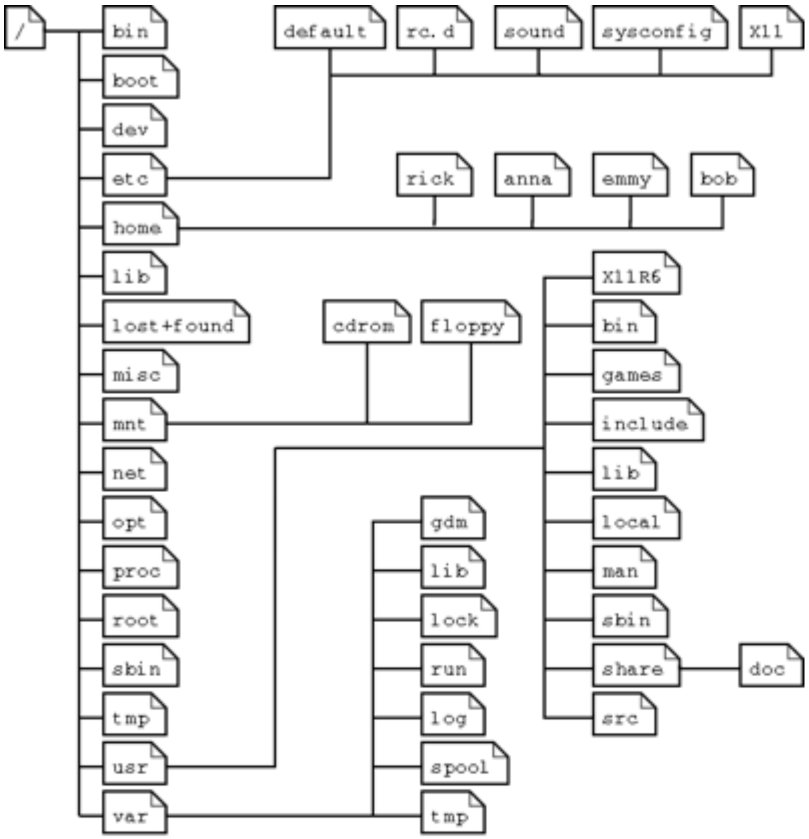
WG 2.8 Kefalonia





High-level languages offer a rich set of tools for organizing, accessing, and modifying memory:

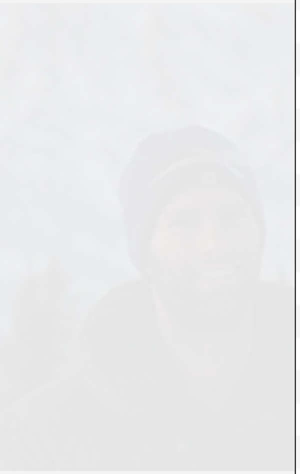
- Data types
- Concurrency models
- Memory models

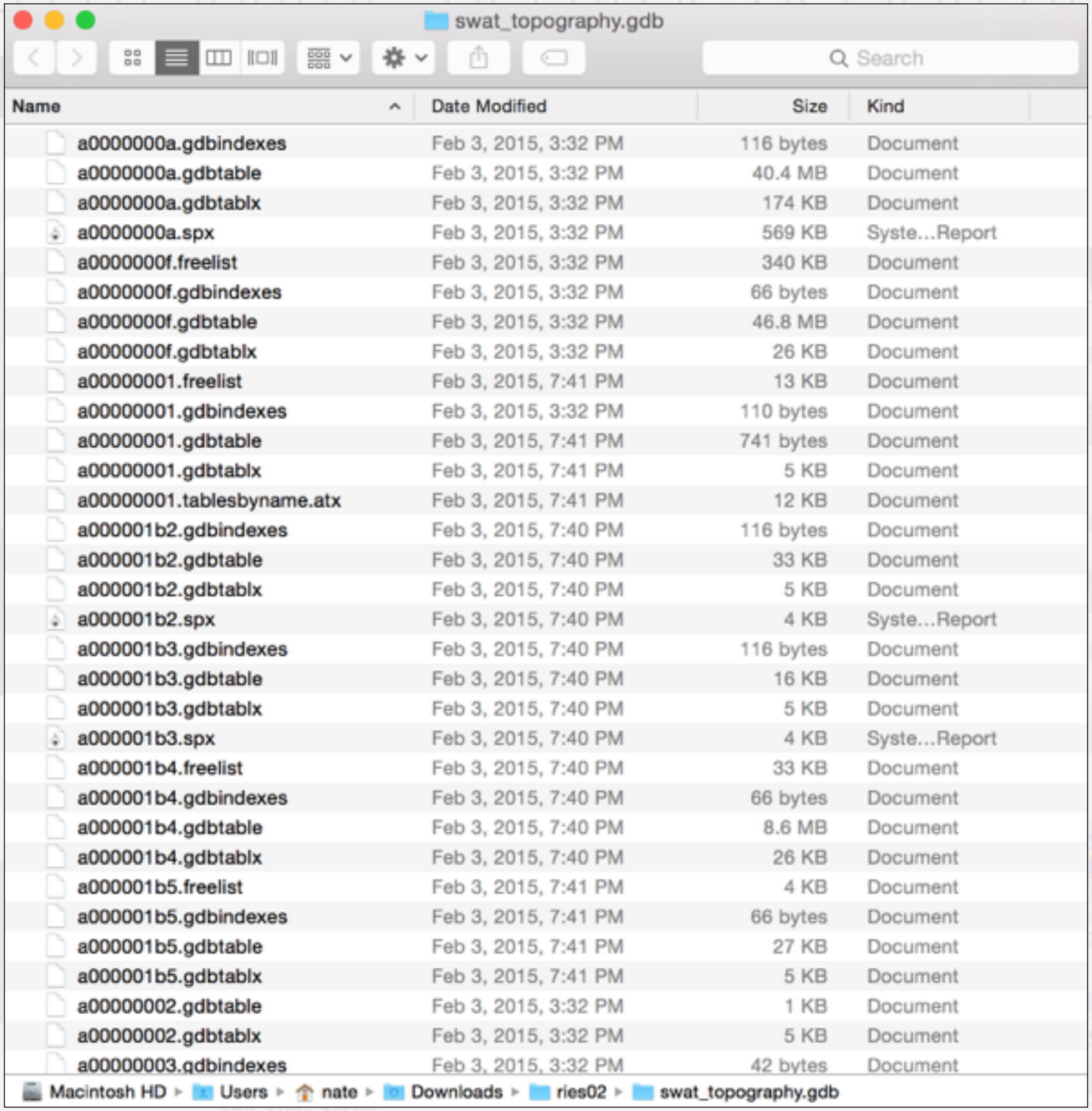


... and a single abstraction for persistent data:
the file system (e.g., with POSIX semantics)

This is a shame, because persistent

ad us!

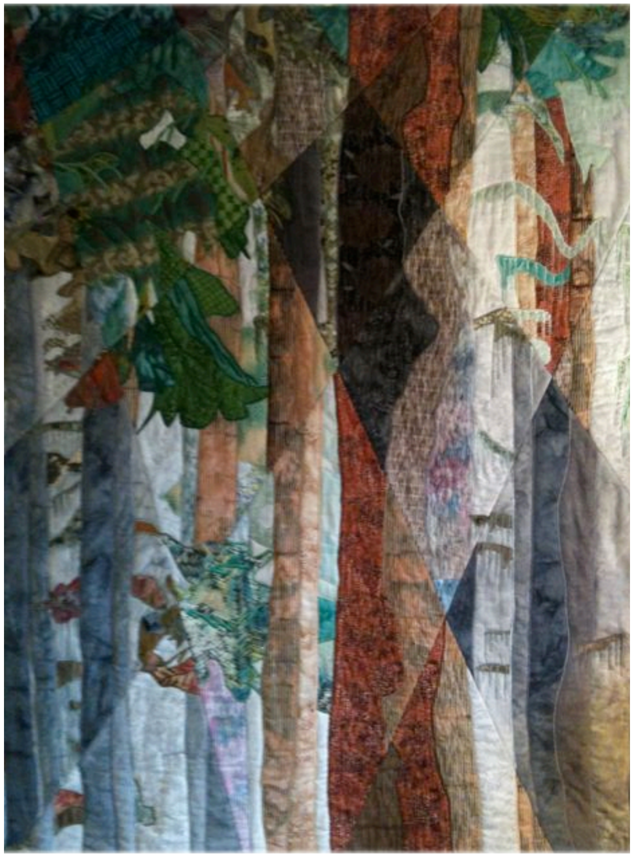




| Name | Date Modified | Size | Kind |
|---|---|---|---|
| a0000000a.gdbindexes | Feb 3, 2015, 3:32 PM | 116 bytes | Document |
| a0000000a.gdbtable | Feb 3, 2015, 3:32 PM | 40.4 MB | Document |
| a0000000a.gdbtablx | Feb 3, 2015, 3:32 PM | 174 KB | Document |
| a0000000a.spx | Feb 3, 2015, 3:32 PM | 569 KB | Syste...Report |
| a0000000f.freelist | Feb 3, 2015, 3:32 PM | 340 KB | Document |
| a0000000f.gdbindexes | Feb 3, 2015, 3:32 PM | 66 bytes | Document |
| a0000000f.gdbtable | Feb 3, 2015, 3:32 PM | 46.8 MB | Document |
| a0000000f.gdbtablx | Feb 3, 2015, 3:32 PM | 26 KB | Document |
| a00000001.freelist | Feb 3, 2015, 7:41 PM | 13 KB | Document |
| a00000001.gdbindexes | Feb 3, 2015, 3:32 PM | 110 bytes | Document |
| a00000001.gdbtable | Feb 3, 2015, 7:41 PM | 741 bytes | Document |
| a00000001.gdbtablx | Feb 3, 2015, 7:41 PM | 5 KB | Document |
| a00000001.tablesbyname.atx | Feb 3, 2015, 7:41 PM | 12 KB | Document |
| a000001b2.gdbindexes | Feb 3, 2015, 7:40 PM | 116 bytes | Document |
| a000001b2.gdbtable | Feb 3, 2015, 7:40 PM | 33 KB | Document |
| a000001b2.gdbtablx | Feb 3, 2015, 7:40 PM | 5 KB | Document |
| a000001b2.spx | Feb 3, 2015, 7:40 PM | 4 KB | Syste...Report |
| a000001b3.gdbindexes | Feb 3, 2015, 7:40 PM | 116 bytes | Document |
| a000001b3.gdbtable | Feb 3, 2015, 7:40 PM | 16 KB | Document |
| a000001b3.gdbtablx | Feb 3, 2015, 7:40 PM | 5 KB | Document |
| a000001b3.spx | Feb 3, 2015, 7:40 PM | 4 KB | Syste...Report |
| a000001b4.freelist | Feb 3, 2015, 7:40 PM | 33 KB | Document |
| a000001b4.gdbindexes | Feb 3, 2015, 7:40 PM | 66 bytes | Document |
| a000001b4.gdbtable | Feb 3, 2015, 7:40 PM | 8.6 MB | Document |
| a000001b4.gdbtablx | Feb 3, 2015, 7:40 PM | 26 KB | Document |
| a000001b5.freelist | Feb 3, 2015, 7:41 PM | 4 KB | Document |
| a000001b5.gdbindexes | Feb 3, 2015, 7:41 PM | 66 bytes | Document |
| a000001b5.gdbtable | Feb 3, 2015, 7:41 PM | 27 KB | Document |
| a000001b5.gdbtablx | Feb 3, 2015, 7:41 PM | 5 KB | Document |
| a00000002.gdbtable | Feb 3, 2015, 3:32 PM | 1 KB | Document |
| a00000002.gdbtablx | Feb 3, 2015, 3:32 PM | 5 KB | Document |
| a00000003.gdbindexes | Feb 3, 2015, 3:32 PM | 42 bytes | Document |

Macintosh HD ▸ Users ▸ nate ▸ Downloads ▸ ries02 ▸ swat_topography.gdb

# The Forest Langauge [ICFP '11]



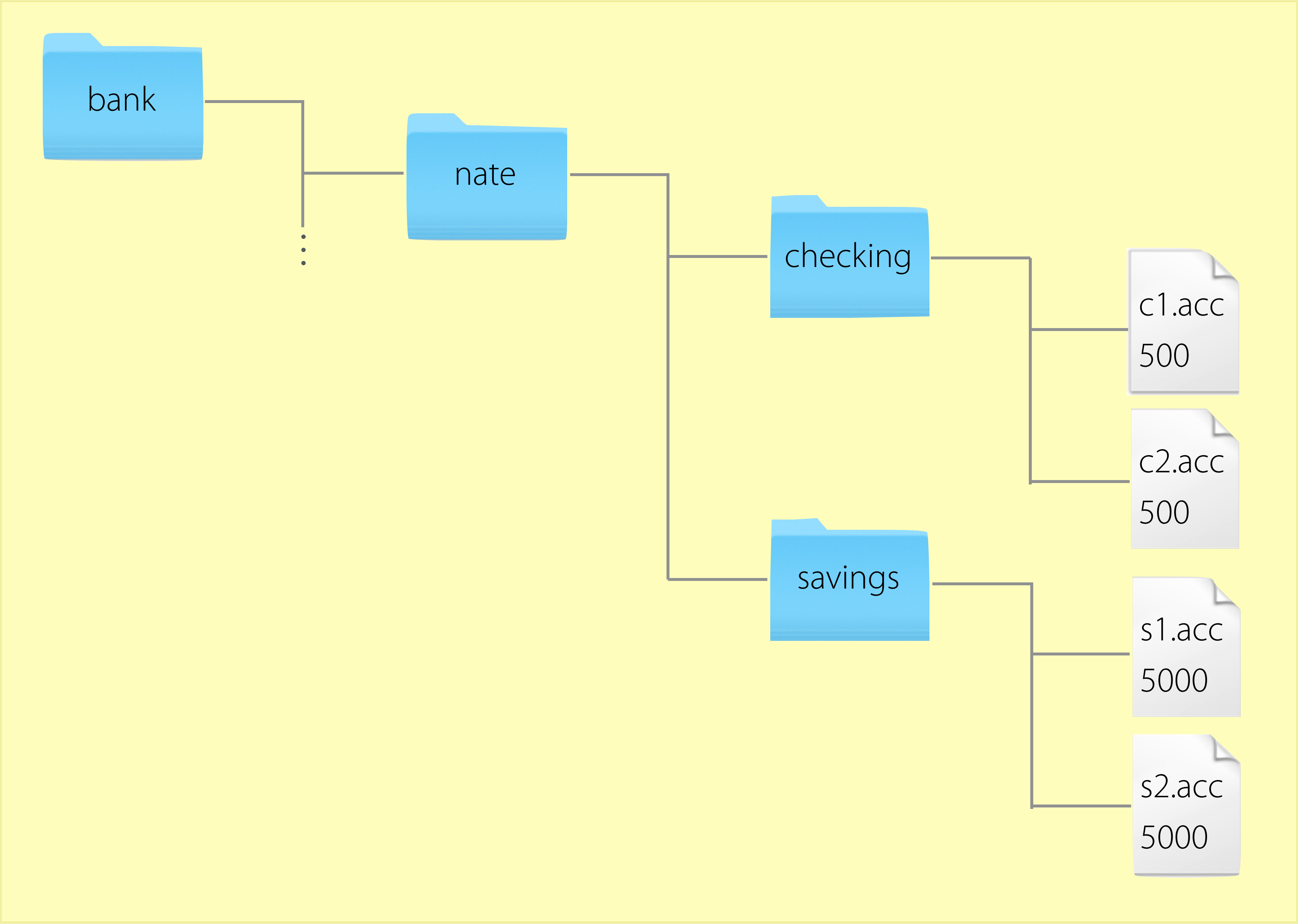
A Haskell DSL for describing and manipulating file stores

Given a Forest specification, the compiler generates

- In-memory representation
- Load and store functions
- Generic programming interface

Describes data "as it is" and *not* as we'd like it to be!

# Example: “Beautiful” Bank Accounts



# Example: Accounts

```
[forest|
   data Bank = Directory {
    clients is Map [c :: Client | c <- matches (GL "*") ]
  }
   data Client = Directory {
     savings :: Accounts
   , checking :: Accounts
   }
   data Accounts = Map [
     acc :: Account | acc <- matches (GL "*.acc")
  ]
   data Account = File AccInfo
|]

[pads|
   data AccInfo = AccInfo { accBalance :: Int }
|]
```

# Forest Artifacts

```
data Bank = Directory { clients :: Map String Client }
data Client = Client { savings :: Accounts, checking :: Accounts}
data Account = Account (File AccInfo)
data Accounts = Accounts (Map String Account)
data AccInfo = AccInfo { accBalance :: Int }

bank_load :: FilePath -> IO (Bank, Bank_md)
client_load :: FilePath -> IO (Client, Client_md)
accounts_load :: FilePath -> IO (Accounts, Accounts_md)
account_load :: FilePath -> IO (Accounts, Accounts_md)

bank_manifest :: (Bank, Bank_md) -> IO Manifest
client_manifest :: (Client, Client_md) -> IO Manifest
accounts_manifest :: (Accounts, Accounts_md) -> IO Manifest
account_manifest :: (Account, Account_md) -> IO Manifest

store :: FilePath -> Manifest -> IO ()
```

Metadata declarations elided for simplicity...

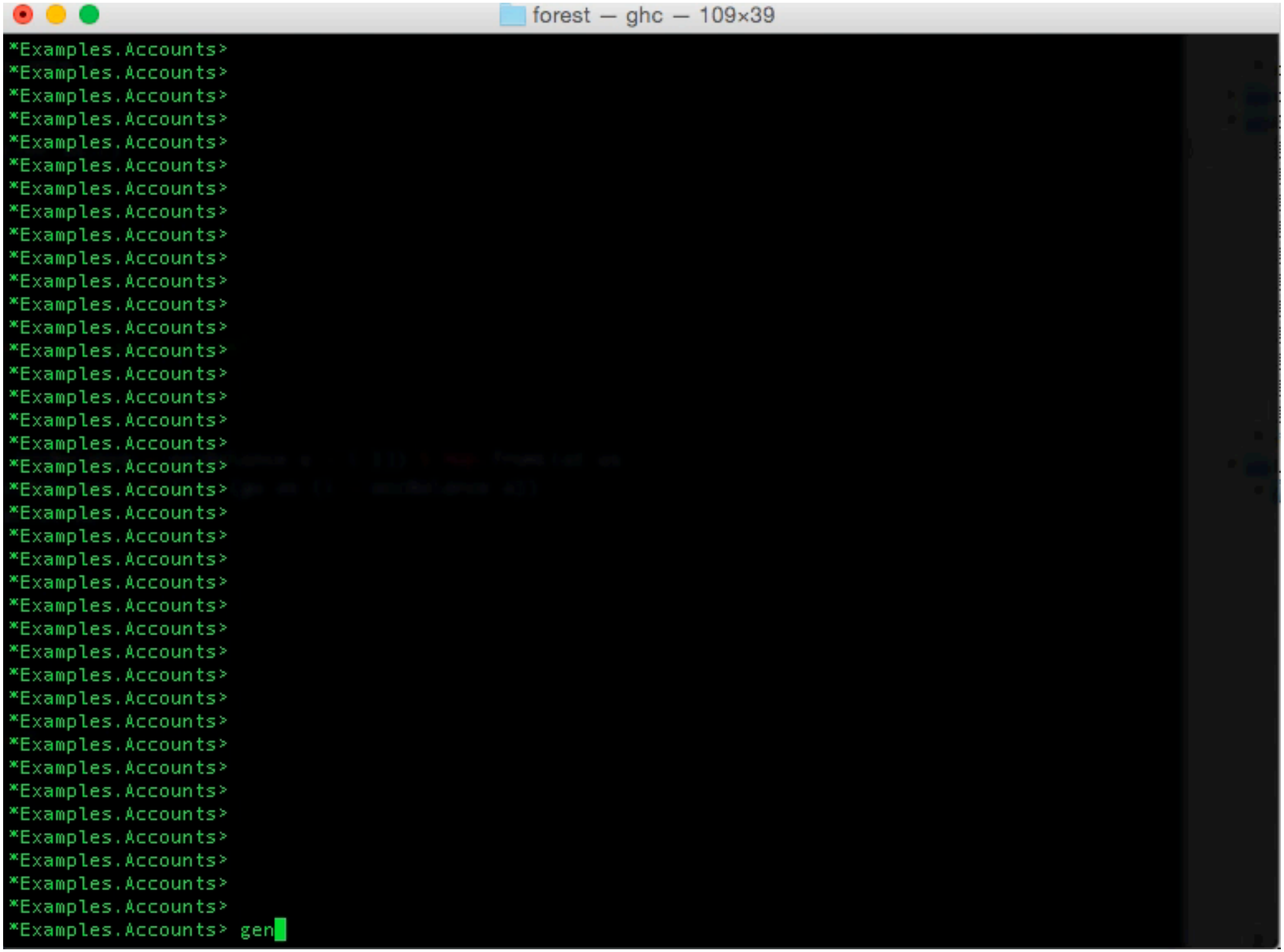
# Example: Accounts

```
balance :: String -> IO Int
balance = do
   (bank :: Bank,_) <- load "/bank"
   return $ tally ((clients bank) ! "nate")

tally :: Data a => a -> Int
tally = everything (+) (mkQ 0 accBalance)

main = balance >>= print

genBank :: IO ()
genBank = ...
```

```
Examples.Accounts> genBank >> main
11000
```

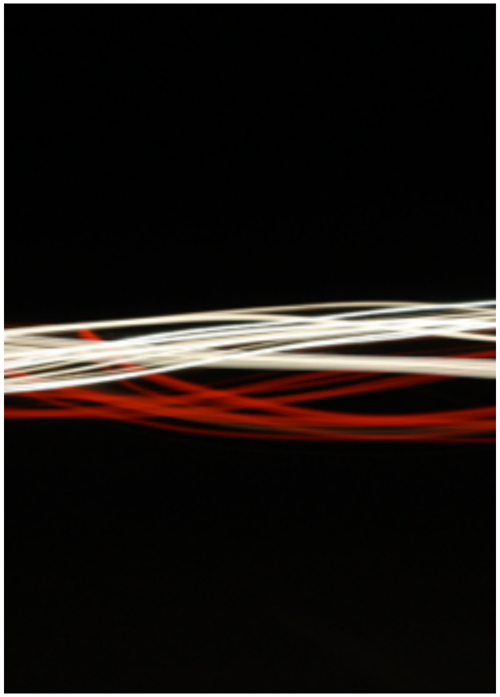
# Example: Accounts

```
withdraw :: String -> Int -> IO ()
withdraw clientid amount = do
   (Bank clients,bank_md) <- load "/bank"
   let n = clients ! "nate"
   let chk,svg = checking n, savings n
   (svg',chk') <- transfer chk svg (amount - min (tally chk) amount)
   chk'' <- reallyWithdraw chk' amount
   let clients' = Map.insert "nate"
     (c { savings = svg', checking = chk'' }) clients
   store (Bank clients',bank_md)

transfer :: Account -> Account -> Int -> IO (Account, Account)
transfer from to amount = ...

main = race_
   (forever $ balance >>= print)
   (forever $ withdraw 200)

genBank >> main
```



```
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts>
*Examples.Accounts> gen
```

# Transactional Forest

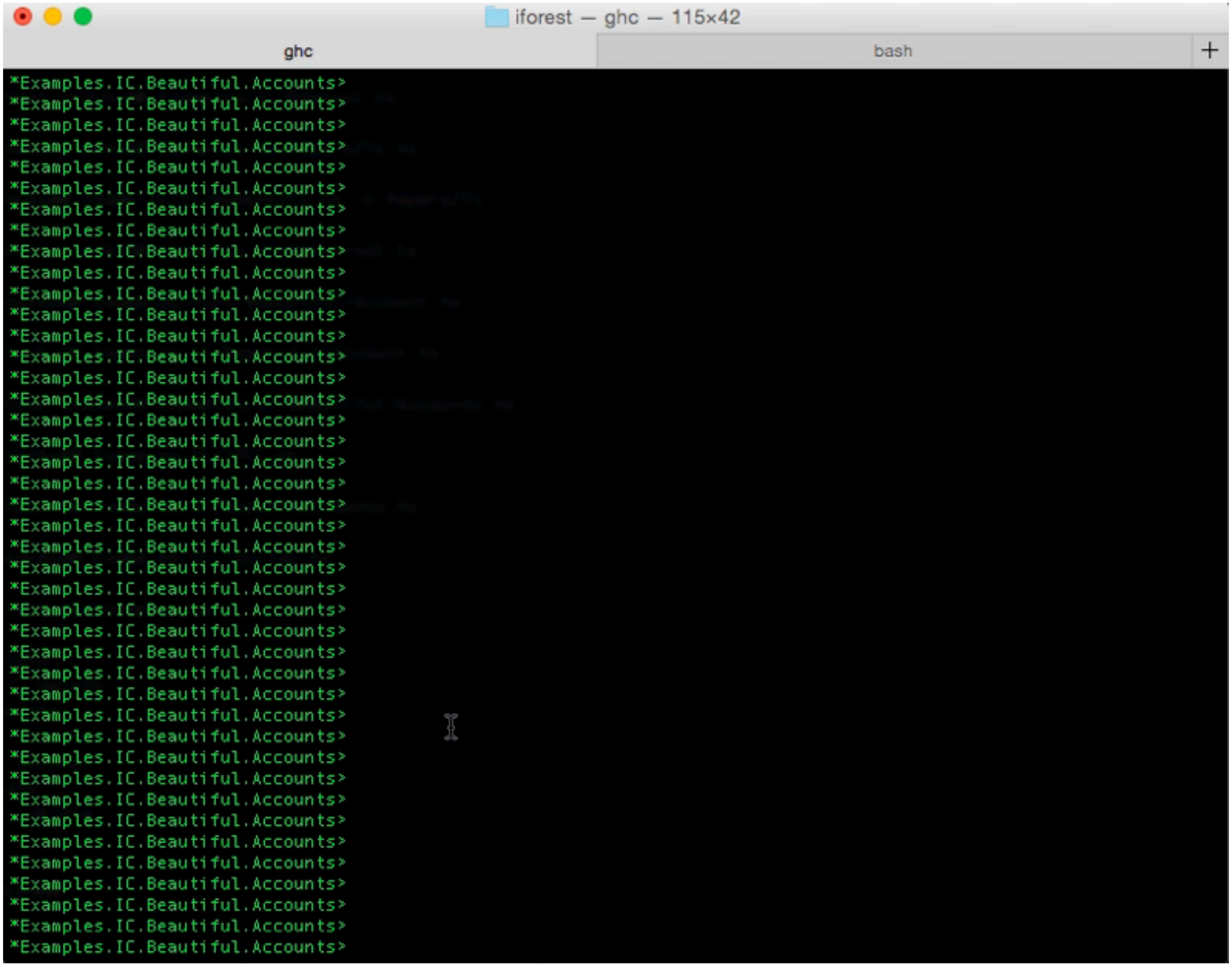


- Provide strong consistency guarantees (serializability)
- Develop novel concurrency control algorithms
- Design rigorous semantics of file and storage systems

```
data FTM a
atomically :: FTM a -> IO a

-- For each Forest description with rep r and metadata m
data FVar r m

new :: FilePath -> FTM (FVar r m)
read :: FVar r m -> FTM (r,m)
write :: FVar r m -> (r, m) -> FTM ()
```

# Example: Transactional Accounts

```
bankClient = do
   bank :: Bank <- new "bank"
   liftM ((!"nate") . clients) (read bank)

balance :: FTM Int
balance = bankClient >>= tally


...


main = race_
   (forever $ atomically balance >>= print)
   (forever $ atomically (withdraw 200))
```



```
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
*Examples.IC.Beautiful.Accounts>
```

# Optimistic Implementation

- Modify standard file system operations to work with a log:
  - Writes modify the log
  - Reads check the log, then the file system
- Upon commit, lock files and validate against writes performed by other threads executing concurrently
- Either abort the transaction or write the effects to the file system

# IMPOSIX Formalization

## Syntax

```
F ∈ File Store
H ∈ Thread-local Heap
M ∈ Thread Metadata
e ::= x
    | open e
    | close e
    | read e
    | write e
    | flock e
    | ...
c ::= skip
    | x := e
    | c1; c2
    | if e then c1 else c2
    | while e do c
    | atomic c
T ::= ⦃<H,M,c1>,..., <H,M,ck>⦄
```

## Semantics

`<F,T> → <F’,T’>`

## Instrumentation

`⟦-⟧ ∈ Com → Com`

Result does not contain any occurrences of **`atomic c`**

## Property

Every compiled concurrent execution equivalent to some serial execution

# A Fly in the Ointment...

The standard optimistic implementation works, provided every thread is managed by Forest...

... but in the presence of non-Forest concurrent threads, serializability can be violated ☹

Standard POSIX operations like `lockf` and `fcntl` operations are not sufficient

# Other Implementations

- **Locking-Based Schemes**

  Enforce exclusive access to files read and written by a Forest transaction

- **Homeostasis Protocol** [SIGMOD '15]

  Analyze Forest descriptions and synthesize custom concurrency control protocols

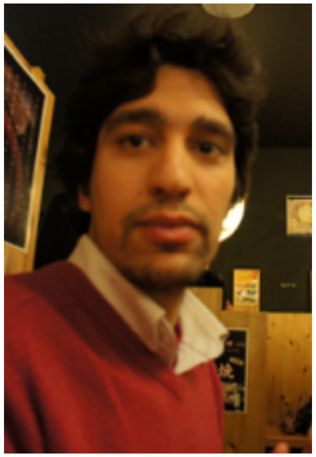
- **Warranties** [NSDI '15]

  Use "semantic leases" to enforce consistency

- **Non-POSIX Alternatives**

  Build on file (or storage) systems with different sets of primitives and semantics

# Thank You!



**http://forestproj.org**