Zipper-based File/OS[1]

An extra-program demo at the Haskell Workshop 2005

We present a file server/OS where threading and exceptions are all realized via delimited continuations.

There are no unsafe operations, no GHC let alone Unix threads, no concurrency problems. Our threads cannot even do `IO` and cannot mutate any global state — and the type system sees to it.

[1] I am very grateful to the Program Committee of the Haskell Workshop 2005 and its chair, Daan Leijen, for the opportunity to present ZipperFS as an extra talk. Many thanks to Chung-chieh Shan for discussions and invaluable help with the presentation.

Getting the first impression

- Load `ZFS.hs` into `GHCi`
- Start up the system: `main` at the `GHCi` prompt
- From some terminal: `telnet localhost 1503`
  - `ls`
  - `cat fl1`
  - `cd d1`
  - `ls`
  - `ls d11`
  - `ls d11/d111`
    That was an empty directory. This all looked like UnixFS. However, there are no `.` and `..`
  - `ls ../d2` – another empty dir
  - `cat ../d2/../fl2`
    Absolute paths work too.

Filesystem term

```
type FileName = String
type FileCont = String – File content
data Term = File FileCont | Folder (Map.Map FileName Term)

data NavigateDir =
  Update Term | DownTo FileName | Up | Next

traverse :: Monad m => (Path → Term → m NavigateDir)
  → Term → m (Term, UpdateStatus)
```

The 'file system' we've just seen is a zipper over a recursive data type based on `Data.Map` of this structure: `Term`. `NavigateDir` defines movement from one subterm to another.

The user defines a traversal function of this signature. It is a `mapM` over the term, in an *arbitrary* monad. It is *not* a fold — merely a map. It pays to define the function that maximally preserves sharing: see `ZipperM.hs`.

Generic Zipper

```
data DZipper m =
    DZipper{
    dz_path :: [FileName],
    dz_term :: Term,
    dz_k    :: NavigateDir → CC PP m (DZipper m)
    }
  | DZipDone Term
```

Once we have defined `traverse` (see `ZipperM.hs`), we can make a zipper. A zipper is an *update* cursor into an immutable data structure — term. It is generic — it depends only on the *interface* of the traversal function, as we shall see shortly. Unlike Huet's Zipper, our zipper is independent of the data structure.

Creating generic zipper

```
pz :: Typeable1 m => Prompt PP m (DZipper m)
pz = pp

dzip'term :: (Typeable1 m, Monad m) =>
  Term → CC PP m (DZipper m)
dzip'term term =  pushPrompt pz
    (traverse tf term »= return . DZipDone . fst)
 where
 tf path term = shiftOP pz
    (\k → return $ DZipper (dir_path path) term k)
 dir_path = foldr filterfn []
 filterfn (PathName fname) acc = fname : acc
 filterfn _                acc = acc
```

We create the zipper via the following generic function. The function is quite short and fits on one slide.

Creating generic zipper (cont)

The function `dzip'term` relies on the delimited continuation operators from the delimcc monad transformer library. Not surprisingly, because zipper is the manifestation of a delimited continuation reified as a `DZipper` record. The zipper maintains the path to the current location in the term. Again, we do so generically, regardless of the term.

Built-in traversal

- `cd /d2`
- `next`
  a few times. Watch for the changes in the “shell prompt”
- When in File, one can do `ls`: indeed, one can `cd` into a file. `cat` is the same as `ls`: both list directories and files.
- a few more `next`
  when the traversal is finished, we are stuck at the root

Unlike UnixFS, our file system has a built-in traversal facility: from *each* node, we can get to the next. Furthermore, our traversal can start from any arbitrary node in the tree.

Multi-threading

- From another terminal: `telnet localhost 1503`
- Enter at the command prompt: `ls`, `cd d1`, `ls`
- Enter `ls` in the first terminal window

We have what looks like multi-threading. However, the whole server is a single Unix process, a single Unix thread and a single GHC thread.

We do not use handles; rather, we read/write sockets directly and rely on `select`.

Our “file server” is an OS, complete with the main `osloop`, “interrupt” handler and and the syscall interface.

We use delimited continuations to implement our processes.

Transactional semantics

- From the first terminal
  - `cd /d2`
  - `touch nf`
  - `ls`
  - `echo "new content" > ../d2/n2`
    Error-check does work...
  - `echo "new content" > ../d2/nf`
  - `cat nf`
  - `rm /`
  - `rm ../d2` – can't remove itself or its own parent
  - `rm ../d1`
  - `cd ..`
  - `ls`
    Indeed, `d1` is gone.
- From the second terminal (the current directory was `d1`)
  - `ls`
  - `ls /`
    Directory `d1` still exists

Transactional semantics (cont)

- From the first terminal
  - `commit`
- From the second terminal
  - `ls`
  - `ls /`

    `d1` still exists. If we open the third terminal, we find that `d1` is gone
  - `refresh`
  - `ls`

    And now `d1` is gone
  - `rm /d2`

    we can remove whole directory trees. Oops...
  - `ls`

    it is gone indeed
  - `refresh`

    but we can easily undo that
  - `ls`

    we see that `d2` is back

Transactional semantics (cont)

- Strongest, “repeatable read” isolation mode
- Undo
- Multiple undo and snapshots are possible

We get this all for free, without any extra programming, courtesy of the zipper

Process isolation

```
run'process ::
  (∀ m. (Typeable1 m, Monad m) => CCM m (OSReq m))
  → CCM IO (OSReq IO)
run'process body = pushPrompt pOS body
```

Here is the function to run our "process". The process function, the first argument, does not have the `IO` type.
The base monad type `m` is left polymorphic. Although a process runs eventually in the `IO` monad, the process *cannot* know that and hence cannot do any `IO` action. It must ask the "OS" by sending an `OSReq`. That means, a process function cannot mutate the `World` or any global state, and *the type system checks that*! Because processes cannot interfere with each other and with the OS, there is no need for any thread synchronization, locking, etc. We get the transactional semantics for free.

Processes and OS requests

```
data OSReq m = OSRDone
     | OSRRead (ReadK m)
     | OSRWrite String (UnitK m)
     | OSRTrace String (UnitK m) – so a process can syslog
     | OSRCommit Term (UnitK  m)
     | OSRefresh (FSZipper m → CCM m (OSReq m))

type UnitK m = () → CCM m (OSReq m)
type ReadK m = String → CCM m (OSReq m)

svc req = shiftOP pOS (return . req)
```

Here's the type of `syscalls`: read, write, commit the changes to the file system, refresh, write to the syslog.
The function `svc` is the "supervisor call". A process invokes `svc` to request a service from the "kernel".

Conflict resolution

Since different processes manipulate their own (copy-on-write) terms (i.e., file systems), when processes commit, there may arise conflicts.

One has to implement some conflict resolution — be it versioning, patching, asking for permission for update, etc. In our system, these policies are implemented at the level of the supervisor rather than at the level of a process. Because processes are “pure”, always ask the supervisor for anything, and the supervisor has the view of the global state, the resolution policies become easier to implement.

Smart sharing

- `quit` on all terminals, kill `main`
- At the `GHCi` prompt: `main' fs2`
- From some terminal: `telnet localhost 1503`
  - `ls`
  - `cd d1`
  - `ls`
  - `cd d1`
  - `ls`

```
fs2 = Folder
  (Map.fromList [("d1",fs2), ("fl1", File "File1")])
```

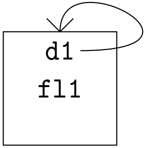
Here is the ‘file system’ `fs2`. It has a cycle: whenever we descend into `d1`, we get back into it. No surprises here: you can do that in Unix, if you are root: hard directory link.

Smart sharing (cont)

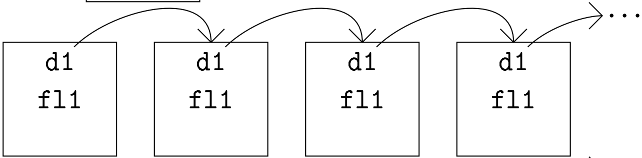
- `touch newfile`
  We are in the directory `/d1/d1/`
- `cd /`
- `ls`
  No `newfile` here!
- `cd d1`
- `ls`
  `newfile` is not here either
- `cd d1`
- `ls`
  Now, it is here
- `cd d1`
- `ls`
- `ls d1`
- `ls d1/d1`
- `ls d1/d1/d1/`

Smart sharing (cont)

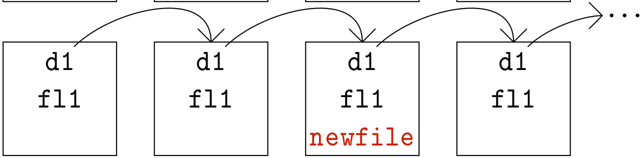
Initially, our file system was like that



However, it *appeared* like this



After we created `/d1/d1/newfile`, it *became*:



When we updated the directory `/d1/d1`, the zipper automatically broke the cycle and introduced three real directories. We get the real copy-on-write. Try to get this with a Unix file system!

Conclusions

Zipper-based file system over any term

- Transactional semantics
- Strongest (repeatable read) isolation mode
- Built-in traversal
- Smart sharing
- Threading and exceptions via delimited continuations
- Static guarantee of processes' non-interference

Future work

- FUSE or NFS or 9P server
- Semantically richer terms: extended attributes, ...
- `cd` into a λ-term in `bash`

Delimited continuations do the right thing for free