

Darcs: Distributed Version Management in Haskell

David Roundy

Cornell University

September 30, 2005



Some incomprehensible equations

A classical density functional $F[\rho]$ for the free energy of a fluid is given by

$$F[\rho] = \int d\vec{r} [f_{ideal}(\rho(\vec{r})) + f_{exc}(\bar{\rho}(\vec{r})) + \rho(\vec{r})\xi(\vec{r})] \quad (1)$$

where the weighted density $\bar{\rho}$ is defined by

$$\bar{\rho}(\vec{r}) = \int d\vec{r}' \rho(\vec{r}') W(\vec{r} - \vec{r}') \quad (2)$$

and the correction term $\rho(\vec{r})\xi(\vec{r})$ is determined by

$$\xi(\vec{r}) = - \int d\vec{r}' \rho(\vec{r}') (k_B T C(\vec{r} - \vec{r}') + W(\vec{r} - \vec{r}')) \quad (3)$$

where $C(\Delta\vec{r})$ is the direct correlation function.

$$\frac{\delta^2 F}{\delta\rho(\vec{r})\delta\rho(\vec{r}')} = -k_B T \rho(\vec{r}) [\delta(\vec{r} - \vec{r}') + C(\vec{r} - \vec{r}')] \quad (4)$$



Outline

Introduction to darcs

Ideas behind darcs

What worked and what didn't

A pure functional language for an SCM?

Laziness and unsafeInterleaveIO

Object-oriented-like data structures

QuickCheck

Foreign Function Interface

Efficient string handling

Handles and zlib and threads

Error handling and cleanup

Optimization experiences



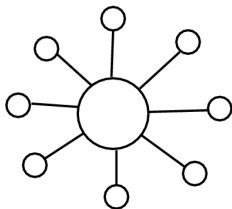
Ideas behind darcs

- ▶ A simple “egalitarian” distributed model
- ▶ “Cherry picking” of changes
- ▶ Avoidance of “merge points”—no history



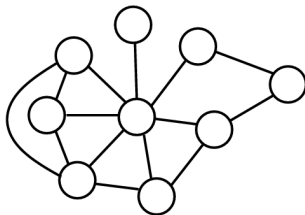
Distributed rather than centralized

Centralized



Examples: CVS, Subversion,
Perforce

Distributed

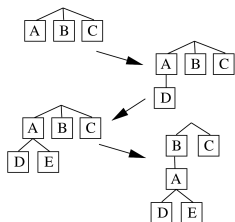


Examples: darcs, Git, Bitkeeper,
monotone, arch

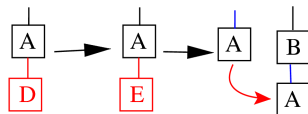


Change-based rather than version-based

Version-based



Change-based



Examples: darcs

Examples: Git, Bitkeeper, Monotone,
CVS, Subversion



Early darcs history

Summer 2002 First version of darcs written in C++

October 21, 2002 First commit of darcs written in Haskell

April 3, 2003 Darcs 0.9.3, its first public release

July 7, 2003 Windows support added



A pure functional language for an SCM?

Large amounts of IO

- ▶ Reading directory trees
- ▶ Reading and writing local files
- ▶ Fetching remote files
- ▶ Running darcs over ssh
- ▶ Sending email

Few “pure” computations

- ▶ Parsing patches
- ▶ Commuting patches
- ▶ Merging patches
- ▶ “Diffing” files



Laziness and unsafeInterleaveIO

Diffing directories mixes IO with “pure” computation

- ▶ Reading directory trees and file contents
- ▶ Computing the “diff” of the two trees

Lazy IO allows us to separate the directory-reading code from the diffing code.



In SlurpDirectory.lhs:

```
slurp :: FilePath -> IO Slurpy
```

```
genslurp_helper :: Bool -> (FilePath -> Bool)  
                -> FilePath -> String -> String  
                -> IO (Maybe Slurpy)
```

```
genslurp_helper usemm nb formerdir fullpath dirname =  
    unsafeInterleaveIO $ do ...
```

In Diff.lhs:

```
smart_diff :: [DarcsFlag] -> (FilePath -> FileType)  
            -> Slurpy -> Slurpy -> Maybe Patch
```



Darcs commands and common code

- ▶ Darcs commands invoked as:

```
darcs get http://darcs.net
```
- ▶ Commands have common behavior:

```
darcs pull --help
```
- ▶ Earlier C++ code used inheritance to handle this.
- ▶ Haskell code uses an object-oriented-like data structure.



DarcsCommand data structure

```
data DarcsCommand = DarcsCommand {  
  command_name :: String,  
  command_darcsoptions :: [DarcsOption],  
  command_command ::  
    [DarcsFlag] -> [String] -> IO (),  
  command_help, command_description :: String,  
  command_extra_args :: Int,  
  command_extra_arg_help :: [String],  
  command_prereq ::  
    [DarcsFlag] -> IO (Either String FilePath),  
  command_get_arg_possibilities :: IO [String],  
  command_argdefaults :: [String] -> IO [String]  
}
```



QuickCheck

QuickCheck makes it easy to create unit tests.

From PatchTest.lhs:

```
prop_readPS_show :: Patch -> Bool
prop_readPS_show p =
  case readPatchPS $ packString $ show p of
    Just (p',_) -> p' == p
    Nothing -> False
```



Foreign Function Interface

The FFI has been crucial in allowing interfaces with existing libraries.

```
foreign import ccall "hscurl.h get_curl"  
  get_curl :: CString -> CString -> CString  
           -> CString -> CInt -> IO CInt
```

```
foreign import ccall unsafe "static zlib.h gzopen"  
  c_gzopen :: CString -> CString -> IO (Ptr ())
```



Efficient string handling

Problem:

- ▶ String as `[Char]` is slow and memory-intensive.
- ▶ `Data.PackedString` still uses 32 bits per character.
- ▶ Would like to pass data to and from C libraries—this rules out use of `UArray`.

Solution:

- ▶ Implement a “PackedString” object holding a `ForeignPtr`.



FastPackedString

```
data PackedString = PS !(ForeignPtr Word8) !Int !Int
```

Allows splitting substrings without copying:

```
takePS :: Int -> PackedString -> PackedString  
takePS n ps@(PS x s _) = if n >= lengthPS ps then ps  
                        else PS x s n
```

```
dropPS  :: Int -> PackedString -> PackedString  
dropPS n ps@(PS x s l)  
  | n >= lengthPS ps = nilPS  
  | otherwise = PS x (s+n) (l-n)
```



FastPackedString

Allows FFI IO:

```
gzWriteFilePS :: FilePath -> PackedString -> IO ()
gzWriteFilePS f (PS x s l) =
    withCString f $ \fstr -> withCString "wb" $ \wb ->
    do gzf <- c_gzopen fstr wb
       when (gzf == nullPtr) $ fail "error"
       lw <- withForeignPtr x $ \p ->
           c_gzwrite gzf (p 'plusPtr' s) l
       when (lw /= l) $ fail $ "error"
       c_gzclose gzf
```



Extending IO

Problem:

How to use shared code to write to a compressed file using zlib or an ordinary file using standard IO.

Constraints:

- ▶ Avoid duplicate code
- ▶ Don't reimplement standard IO
- ▶ Efficiency in time and space
- ▶ Don't know in advance how big the file will be



Attempt #1: Output lazy string

```
gzWriteFile :: FilePath -> String -> IO ()  
gzWriteFile f s =  
  withCString f $ \fstr -> withCString "wb" $ \wb ->  
  do gzf <- c_gzopen fstr wb  
     mapM_ (c_gzputc gzf . ord) s  
     c_gzclose gzf
```

- ▶ Writes one character at a time
- ▶ Consumes a lazy string, for efficient memory use
- ▶ *Very slow*



Attempt #2: write to Handle using forkIO

```
gzOpenFile :: FilePath -> IOMode -> IO Handle
gzOpenFile f WriteMode =
    withCString f $ \fstr -> withCString "wb" $ \wb ->
    do gzf <- c_gzopen fstr wb
       (readfd, writefd) <- createPipe
       readH <- fdToHandle readfd
       forkIO $ gzwriter gzf readH
       fdToHandle writefd
```

- ▶ Create pipe and attach one end to a Handle
- ▶ forkIO a thread to read from other end and write to file
- ▶ Somewhat faster
- ▶ Prone to race conditions when main thread exits before write is complete



Attempt #3: write to Handle through pipe with fork

```
gzOpenFile f WriteMode = withCString f $ \fstr -> do
    fd <- gzwriter fstr
    fdToWriteHandle fd f
```

```
int gzwriter(char *f);
```

- ▶ “gzwriter” is a C function that returns one end of a pipe.
- ▶ “gzwriter” itself calls fork to write compressed data
- ▶ Fast and memory efficient
- ▶ Quite awkward and nonportable



Attempt #4: write through Handle pith pipe and pthreads

```
gzWriteToFile :: FilePath -> (Handle -> IO ()) -> IO ()
gzWriteToFile f job =
  withCString f $ \fstr -> withPthread $ \pth ->
  do fd <- gzwriter fstr pth
     thid <- peek pth
     h <- fdToWriteHandle fd f
     job h
     gz_join thid
```

- ▶ “gzwriter” returns one end of a pipe, and the ID of a thread at the other end of the pipe.
- ▶ Fast and memory efficient
- ▶ Quite awkward, but marginally portable
- ▶ “gzWriteToFile” idiom eliminates race condition



Attempt #5: write lazy list of PackedString

```
gzWriteFilePSs :: FilePath -> [PackedString] -> IO ()
```

- ▶ Full circle: writes a lazy list
- ▶ Simple—to simple for general use
- ▶ Fast and memory efficient
- ▶ But we've given up on using Handle for IO to files.



Error handling and cleanup

Error handling has been a challenge for darcs. Basic error handling consists of using `bracket` or `catch` to create a “withXXX” function:

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

```
withLock :: String -> IO a -> IO a
```

```
withLock s job =  
    bracket (getlock s) releaseLock (\_ -> job)
```

One wants to ensure that the lock is always released, but alas, this is not so easy.



Which bracket or catch?

There are three instances of `bracket` in the standard libraries:

- ▶ `IO.bracket`
- ▶ `System.IO.bracket`
- ▶ `Control.Exception.bracket`

There are two instances of `catch`:

- ▶ `Prelude.catch`
- ▶ `Control.Exception.catch`



Which bracket or catch?

There are three instances of `bracket` in the standard libraries:

- ▶ `IO.bracket`
- ▶ `System.IO.bracket`
- ▶ `Control.Exception.bracket`

There are two instances of `catch`:

- ▶ `Prelude.catch`
- ▶ `Control.Exception.catch`



Control.Exception

Control.Exception handles properly

- ▶ Ordinary exceptions (e.g. fail, IO errors)
- ▶ `exitWith` from within the bracket
- ▶ Asynchronous exceptions

Control.Exception does *not* handle

- ▶ Posix signals (e.g. SIGHUP, SIGINT, SIGKILL)
- ▶ win32 keyboard interrupt

Moral:

Robust cleanup code requires both platform-specific code, which in the case of win32 requires use of the FFI.



Optimization in Haskell

High level: we want laziness

- ▶ Laziness makes possible non-invasive space optimizations. (e.g. lazy reading, parsing and consumption of patches).
- ▶ Space optimizations often *dramatically* improves time performance.
- ▶ Laziness makes it very easy to introduce memory leaks.

Low level: we want strictness

- ▶ Can use the Foreign Function Interface to call C or library functions for low-level operations (e.g. string compare).
- ▶ Optimizing low-level Haskell code possible, but challenging.



Lazy parsing of patches

A single parsed patch can require hundreds of megabytes of memory, so it's helpful to be able to lazily parse a patch (in which case we'll have to call `error` if we have a malformed patch).

```
readPatch :: Stringalike s => s -> Maybe (Patch, s)
readPatch ps = case parse_strictly readPatch' ps of
  Just (Just p, ps') -> Just (p, ps')
  _ -> Nothing
```

```
readPatchLazily :: Stringalike s => s -> (Patch, s)
readPatchLazily ps = case parse_lazily readPatch' ps of
  (Just p, ps') -> (p, ps')
  _ -> impossible
```



Consuming large lazy data

1. Data must be consumed as it is generated (no reverse)
2. Data may only be used once! (no length)

How we satisfy these constraints:

- ▶ If we use a patch multiple times, we must write it to disk and then reread it. (e.g. when recording a patch)
- ▶ Avoid reading whole lists of patches while deciding what to do with them. (this is hard!)



An example low-level optimization

```
linesPS :: PackedString -> [PackedString]
linesPS ps = case wfindPS (c2w '\n') ps of
    Nothing -> [ps]
    Just n -> takePS n ps : linesPS (dropPS (n+1))
{-# INLINE wfindPS #-}
wfindPS :: Word8 -> PackedString -> Maybe Int
wfindPS c (PS fp s l) =
    unsafePerformIO $ withForeignPtr fp $ \p->
    do let p' = p `plusPtr` s
        q <- memchr p' (fromIntegral c) (fromIntegral l)
        return $ if q == nullPtr
            then Nothing
            else Just (q `minusPtr` p')
```



GADTs and patch theory correctness

We can encode some invariants of patch theory in the type system:

```
data Patch a b where
  Nil :: Patch a a
  Hunk :: Int -> ... -> Patch a c
  (:-) :: Patch a b -> Patch b c -> Patch a c
```

```
data CommutePair a c where
  (:<->) :: Patch a b -> Patch b c -> CommutePair a c
commute :: CommutePair a c -> Maybe (CommutePair a c)
```



GADTs and patch theory correctness

Here is a simple example bug which triggers a type error:

```
test (b :- c) = commute (c :<-> b)
```

Quantified type variable 'b' is unified with another quantified type variable a
When trying to generalise the type inferred for 'test'

Signature type: forall a b c.

Patch a b -> Patch b c -> Maybe (Commutepair a c)

Type to generalise: Patch b b -> Patch b b -> Maybe (Commutepair b b)

In the type signature for 'test'

When generalising the type(s) for 'test'



Conclusions

- ▶ Haskell has worked for darcs
- ▶ Laziness allows clean and efficient code, but makes avoidance of memory leaks tricky.
- ▶ Lazy IO opens many cans of worms, but in my judgement is worth it.
- ▶ Error handling is insufficiently supported by the the standard libraries (but with Concurrent Haskell, it *is* sufficiently extensible).
- ▶ I wish IO interface were more extensible (using Handles to write to custom “files”)
- ▶ The Foreign Function Interface is great.

My thanks to the Haskell compiler and library developers for a great tool!

