Darcs: Distributed Version Management in Haskell

David Roundy
Cornell University
droundy@darcs.net

Abstract

A common reaction from people who hear about darcs, the source control system I created, is that it sounds like a great tool, but it is a shame that it is written in Haskell. People think that because darcs is written in Haskell it will be a slow memory hog with very few contributors to the project. I will give a somewhat historical overview of my experiences with the Haskell language, libraries and tools.

I will begin with a brief overview of the darcs advanced revision control system, how it works and how it differs from other version control systems. Then I will go through various problems and successes I have had in using the Haskell language and libraries in darcs, roughly in the order I encountered them. In the process I will give a bit of a tour through the darcs source code. In each case, I will tell about the problem I wanted to solve, what I tried, how it worked, and how it might have worked better (if that is possible).

Categories and Subject Descriptors J.0 [*Computer Applications*]: GENERAL

General Terms Languages

1. Introduction

Darcs is a distributed revision control system[1]. It differs from most other modern revision control systems in that it is "change-oriented" rather than being "version-oriented", which is to say that in darcs the fundamental object that is tracked are changes made, rather than a sequence of states. The change-oriented philosophy of darcs has a number of advantages, but requires a considerable amount of "patch arithmetic" to handle merging and reordering of changes in a lossless manner.

I started writing darcs using C++ in the spring of 2002. Starting in the fall of that year, I rewrote darcs in Haskell. Partly this was because I was sick of C++, and partly because there were many bugs in the existing C++ code that a complete rewrite seemed necessary. I chose Haskell mostly because I did not want to stay with C++, but did want a strongly typed language so that at least some of my mistakes could get caught at compile time. I had heard of Haskell once or twice on slashdot, and it sounded appealing. A bit of experimentation suggested that its syntax and expressiveness were worthwhile.

Darcs is an interesting application for a pure functional programming language, in that it is very IO-intensive, and IO is not commonly thought of as being a strong point of functional languages. On the other hand, much of darcs' code involves the manipulation of patches, which is purely functional code. My experience, however, has been that both sides of darcs have benefited from the choice of Haskell as the programming language.

[1] Revision control systems are also known as "version control systems", "source code management" or "software configuration management"... the acronyms never end. I prefer revision control system, but SCM is more commonly used.

Haskell'05 September 30, 2005, Tallinn, Estonia.
Copyright © 2005 ACM 1-59593-071-X/05/0009...$5.00.

2. Laziness and unsafeInterleaveIO

One of the key operations in darcs that needed to be implemented was a "`diff`" algorithm, which would take two directories and return set of changes describing the difference between them. I knew which algorithms I wanted to use, and just needed to implement them. Unfortunately, there is a lot of tedious directory traversal required. In C++, this directory traversal code had to be interspersed with the `diff` code, since we cannot afford to store the contents of both directory trees in memory.

Haskell allows a nicer approach. I wrote one function to lazily read an entire directory tree into memory (a "`Slurpy`" data type), and another to do the recursive `diff` itself. This required that I learn to use `unsafeInterleaveIO`, which was pretty easy, and resulted in Haskell code which was far cleaner than the earlier C++ code.

This was the first feature of Haskell that struck me as being a major improvement over other languages. By separating execution order from code layout, one is able to write cleaner, more modular code. This should not be news to anyone involved with Haskell, but it is worth reporting that this has indeed been helpful in darcs.

There *is* a limit here, however. Having been so excited about the separation between directory IO and directory manipulation made possible by lazy IO, I went a bit overboard, and did *everything* using pure functions combined with lazy IO. This ended up leading to scenarios where the entire directory tree needed to be held in memory because a function was not sufficiently lazy. Much of the recent optimization work (largely done by Ian Lynagh) has involved switching to more often work directly in the IO monad in order to robustly obtain much more modest memory usage. We were able to retain the cleanliness of code by creating a monad class allowing us to write code that can either be executed in the IO monad or used as a pure function.

3. Object-oriented-like data structures

One feature of darcs that came almost directly from the earlier C++ version was a framework for handling separate subcommands. Darcs is invoked using commands similar (in some ways) to those of CVS, e.g. the command "get" is invokes by

```
  darcs get http://darcs.net
```

In C++, this was handled by an abstract parent class from which were descended one class for each subcommand. The main darcs function then checked the command line against the names of the different subcommands, and the arguments were processed using `getopt` according to the list of legal flags for that subcommand. This code layout translated very naturally into Haskell, `DarcsCommand` being the following 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]
}
```

This framework is even somewhat more natural than it was in C++, since we are not forced to define a separate *type* for each *object*.

4. QuickCheck

One of the problems I had with the initial C++ darcs was that I had no unit testing code. Within two weeks of the first darcs record, I started using `QuickCheck` to test the patch functions, and the same day I fixed a bug that was discovered by `QuickCheck`.

`QuickCheck` makes it very easy to define properties that functions much have, which are then tested with randomly generated data. A simple example is:

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

The trouble with `QuickCheck` is in creating valid patches (and sequences of patches) to use as input. One can define custom generators, but it is hard to determine if a sequence of tests is valid. All too often the bugs found using `QuickCheck` have been bugs in the generation of random patches rather than bugs in darcs itself.

Still, `QuickCheck` has been invaluable in testing darcs as it has moved forward. My one gripe with `QuickCheck` has been that it does not seem to be possible for the code *calling* `QuickCheck` to discover if the test passed or failed.

5. Foreign Function Interface

The Foreign Function Interface (FFI) has been absolutely essential in darcs, and I have very little but good to say about it. My biggest gripe would be that when I was first learning to use it there were so many tools that are layered over it (GreenCard, HDirect, etc) that it was quite a while before I realized how easy the FFI is to use in its raw form. Darcs' first use of the FFI came about while adding support to darcs to use `libcurl` to support `http` downloads. This feature that turned out to be quite easy to add. Most FFI imports in darcs are as simple as defining

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

and using `withCString` to convert Haskell Strings into `CStrings`.

6. Efficient string handling

In the beginning, darcs used `String` to handle file contents. Eventually I realized that this required too much memory and was too slow, so I switched to `PackedString`. This lead to a major improvement in speed, but still required four bytes per character, since it worked with Unicode characters. However, the IO routines darcs uses guarantee that each character will contain only a single byte. So after some frustration I implemented my own version of `PackedString` called `FastPackedString`.

The original version of `FastPackedString` was implemented using `UArrays` to store the characters. Besides using one quarter the memory, the `FastPackedString` allowed the splitting of a string without copying memory, although this causes the original string to be held in memory, which *could* be problematic. This feature allows darcs to split a file into lines and store both the original file and the split file at only the cost of the locations of line endings.

Both memory use and file access speed remained a problem, and I decided to try using `mmap` to read file contents. So I rewrote `FastPackedString` using a `ForeignPtr` to store the actual data. Interestingly just this conversion gave a 15% speedup—suggesting a problem either with the efficiency of `UArray` or with my use of it. The `ForeignPtr` storage allowed me to call optimized C library calls such as `memcmp` to efficiently perform certain `FastPackedString` functions, such as `(==)`, which were bottlenecks. This is another case of the usefulness of the FFI.

One thing lead to another, and soon I was doing most of my optimization within `FastPackedString` by writing fast C routines that were then called from Haskell. This says good things about the FFI, in that it allowed me to easily write hand-tuned code in C to optimize key functions, but it is less good news that I found this so much easier than writing efficient functions in Haskell.

It would be nice to be able to access a particular chunk of memory both as a `ForeignPtr` and as a `UArray`. This would require that the memory will not be modified by the `ForeignPtr` calls, so it would be an "unsafe" function. But when I know that a chunk of memory is not going to be modified, I would prefer access it with either C code or pure Haskell code, rather than `IO` code using `Ptrs`. This illustrates a limitation of the FFI, which is that it cannot interact in a friendly way with Haskell data structures except by copying. Thus, if I want to be able to use C library functions with a chunk of memory I have to go all the way and store the data in a `ForeignPtr`, which eliminates the possibility of *also* accessing it as pure Haskell data.

7. Handles and zlib and threads

The first approach to writing compressed files in darcs using zlib was based on a simple function which wrote the file one character at a time:

```
  gzWriteFile :: FilePath -> String -> IO ()
```

This is memory-efficient as long as the `String` is generated lazily, and it was pretty simple to write using the FFI, but was horribly slow. Making one library call per character is a bad idea in any language, but is particularly painful in Haskell.

So I decided to write a function that would open a compressed file for writing, and return a `Handle` so I could then use the same patch-writing code for writing to either compressed or uncompressed files. This would be hard to do in C, but it seemed like in a functional language like Haskell it should not be a problem. It turned out to be very problematic indeed.

My first attempt was under a hundred lines of pure (concurrent) Haskell. It created a pipe and used `ForkIO` to generate a thread reading from one end and writing to the compressed file, while the other end of the pipe was attached to the `Handle` to which we wish to write. It seemed like an elegant solution, but there was a race condition that caused trouble if darcs exited before the spawned thread finished writing to disk.

The second attempt used the FFI and `fork` to spawn an OS process from C, which read from one end of a pipe and wrote to disk. This code was buggy (not to mention complex), and I soon switched to using `pthreads` to spawn an OS thread to read from the pipe and write compressed data to disk. This worked, and was race-free, but was a continual portability problem. There is a pthreads library available on windows, so we had an efficient cross-platform solution. However, the use of pthreads caused more users to have trouble compiling darcs than anything else.

Eventually, we moved back to a function quite similar to the original function that wrote a lazy `String`:

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

This function differs from the original `gzWriteFile` in that it writes a whole block of data with a single FFI call rather than one character at a time. This makes a huge difference in performance. The process of creating such a `[PackedString]` is nicely handled by "`Printer`", a formatting module by Ian Lynagh.

It would be *very* nice if the standard libraries were more extensible. This is another case where the FFI is helpful, but we are forced to choose between using the FFI to get something done and using the standard Haskell facilities—the `Handle`-based IO routines.

8. Starting other processes

There are numerous instances in which darcs needs to execute an external program. Examples include ssh, a text editor or sendmail. In some cases, such as ssh, we would like to provide the input, and capture the output for display to the user. At first, all external programs were started by calls to `system`. This is a fragile way of starting a program, since any shell meta-characters must be escaped. We may be able to do better with `rawsystem`, but we still would need to be careful when passing arguments that contain spaces. In general, it would be preferable (on POSIX systems) to start external programs with `fork` and `execvp`.

We had major difficulties with a "virtual timer expired" error, and eventually found that we had to turn off the VT_ALARM after forking and before execing—the solution was found in the GHC source code to `system` and friends. This is fundamental problem, but illustrates the point that as wonderful as the FFI is, there *are* pitfalls that can cause serious trouble.

In this particular case, hopefully the new `System.Process` module will prove helpful. We have not yet started using it, mostly because we still want compatibility with older versions of GHC.

9. Error handling and cleanup

An issue that persisted for quite a while were problems with failing to clean up properly when darcs is interrupted. The function that one would think to use for this purpose is `bracket`

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

which allows one to perform an initialization, run a calculation and then afterwards clean up, with the cleanup being performed even if the function throws an exception. The Haskell standard library has no less than three separate versions of `bracket` (two of which are identical). The key is to use `Control.Exception.bracket`, which causes the cleanup function to be run even if the code exits with `exitWith` or if it receives an asynchronous exception.

Additional confusion results from the fact that POSIX signals still cause a program to die without running the cleanup. We dealt with this problem by introducing on POSIX systems a signal handler that throws an asynchronous exception when a signal is received. On Windows, we use the FFI to call `setConsoleCtrlHandler` to achieve a similar effect.

None of this was prohibitively hard, but it should be easier to write a robust IO function that creates a temporary file and then removes that file when it is finished.

A very useful idiom for this sort of function is the "withSomething" idiom, which shows up scattered across the standard libraries. The idea is to write a function such as

```
withSomething :: XXX -> (b -> IO a) -> IO a
```

where `XXX` is some appropriate input that allows you to create an object of type `b` that involves a resource that needs to be freed when the function is complete. These functions are most often implemented with `bracket`. A few examples of this idiom in darcs are

```
withSignalsHandled :: IO a -> IO a
withRepoLock :: IO a -> IO a
withLock :: String -> IO a -> IO a
withTemp :: (String -> IO a) -> IO a
withOpenTemp :: ((Handle, String) -> IO a) -> IO a
withTempDir :: String -> (String -> IO a) -> IO a
withNamedTemp :: String -> (String -> IO a) -> IO a
```

One of the keys to writing of robust code is writing functions that cannot be easily misused, and this is one area where I feel Haskell is particularly strong. As discussed above, it has taken some work to write a robust and correct `withLock` function, but once that function has been properly written, it is almost impossible to use that function in such a way that a lock file is left behind when darcs exits (the exception being cases such as `kill -9` or reckless use of the FFI).

10. Optimization experiences

My experiences optimizing darcs have been mixed. Optimizing Haskell code seems to usually boil down to making the code either more strict or more lazy. Increasing laziness is often helpful in reducing memory usage, while increasing strictness in low-level functions usually makes them faster. The trouble is that it is not always easy to tell which category a function falls into—and it is rarely obvious how a given change will affect the laziness of a function. Profiling has been a very helpful tool when optimizing, although sometimes the profiling itself changes the program's timing behavior.

I have had more success with time optimizations than memory optimizations, although Ian Lynagh has been very successful with the latter. Time optimization most often consist of working on the lowest level functions, which are called in the innermost loops. In many cases—particularly in `FastPackedString`, where darcs spends much of its time—optimization has consisted of rewriting a key function in C or calling a C library function, having chosen that function on the basis profiling. At a higher level, one can often rewrite a function so that it calls more efficient lower-level functions, as was the case with `gzWriteFile` and `gzWriteFilePSs`. In both cases, optimization is reasonably straightforward.

At higher levels, we more often want laziness than strictness—since lazy evaluation at the highest level costs very little in time, but improves the memory usage and consequently garbage-collecting efficiency and locality of access to memory. Ian has made a number of improvemnts in the memory efficiency of darcs. Most of his improvements have revolved around arranging to never hold an entire parsed patch in memory, but instead to consume the patch as we lazily parse it. However, relatively subtle changes can have disastrous effects by causing a patch to be retained in memory.

Conclusion

Darcs has been a highly successful project. It has grown far faster than I ever imagined—largely through the contributions of some very skilled programmers, but also because Haskell itself allows the creation of clean internal interfaces in the code, so contributors working on one feature do not need to learn even the *flow* of the entire code. Although we have had efficiency problems in the past, I hope that darcs will soon be a demonstration that Haskell code need not be inefficient.