Diff and merge

COMP 150 - Applied Functional Programming

November 19, 2012

Review

Let's work with the following subset of Darcs patches:

```
data Patch = AtPath Path PatchAction
           | Atomic [Patch] -- sequence of patches bundled together

data PatchAction = RemoveEmptyFile
                 | CreateEmptyFile
                 | ChangeHunk { offset :: Int      -- starting line number
                              , old    :: [String] -- list of old lines
                              , new    :: [String] -- list of new lines
                              }
```

On November 2, we developed a function for commuting `ChangeHunk` patches. Today we'll take a closer look at where patches come from (diff) and what to do with multiple lines of development (merge).

Computing patches from files (diff)

Allow me to define a little language of file-editing commands:

```
data Edit = C         -- copy the current input line to the output
          | I String  -- insert the argument line into the output
          | D String  -- delete the current input line, which must match
```

We'll use this little language to get to `diff`.

(1) Write a function that uses a list of edits to convert one file to another:

```
applyEdits :: [Edit] -> [String] -> Maybe [String]
```

Hint: produce a list of type `[Maybe String]` and then apply `sequence` to it.

(2) Write a function

```
editsToPatch :: [Edit] -> Path -> Patch
```

which produces an `Atomic` list of hunk patches. The list should be as short as possible.

(3) Write a function

```
diff :: [String] -> [String] -> [Edit]
```

such that

```
applyEdits (diff f1 f2) f1 == Just f2
```

Make sure that the output of `diff` has as many `C` edits as possible (or as few `I` and `D` edits as possible).

Hints:

- You can write `diff` as a simple recursive function.
- Try case analysis on the *output* of `diff`.
- You can make `diff` efficient by using Luke Palmer's `data-memocombinators` package.

Simple merge

Define an interval as

```
data Interval = Interval { start :: Int, length :: Int }
```

And define

```
follows, disjoint :: Interval -> Interval -> Bool
follows late early = start late >= start early + length early
disjoint i1 i2 = (i1 `follows` i2) || (i2 `follows` i1)
```

Define the "affected interval" of a single hunk patch as

```
affectedInterval :: PatchAction -> Interval
affectedInterval patch = Interval (offset patch) (length (old patch))
```

(4) Write a function

```
mergeHunk :: PatchAction -> PatchAction -> Maybe (Pair (Pair PatchAction))
```

such that if

```
disjoint (affectedInterval p) (affectedInterval q)
```

then

```
mergeHunk p q = Just ((p, q'), (q, p'))
```

where

```
old p' == old p                old q' == old q
new p' == new p                new q' == new q
```

and the patch sequence `p; q'` is equivalent to `p'; q`.

(5) Write a more general merge function that can merge two *lists* of `PatchAction`s on the same file. Figure out what type you'd like to give it.

(6) Google the phrase "git is inconsistent" and find the blog post at http://r6.ca/blog. Using your `diff` and `merge` functions, find out if your merge is consistent or inconsistent.

Bonus questions

(7) Write Quickcheck properties can you write for `applyEdits`.

(8) Generalize this code to work on our model repository, and compare it with git's merge algorithm.