Universidade do Minho

Department of Informatics

*Formal Methods in Software Engineering*

# Analyzing and Improving Darcs Quality

Iago Abal

iago.abal@gmail.com

February 23, 2011

# Maintainability Analysis of Darcs 2.5

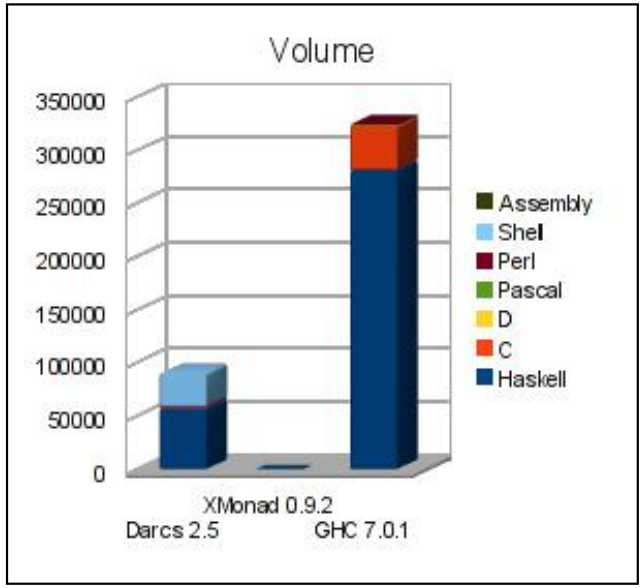
1. Volume
2. Complexity per Unit
3. Code duplication
4. Unit size
5. Module coupling
6. Unit testing
7. Maintainability rating

# Overview

- Functional code metrics?
- Self developed code for take metrics.
  - Parsing: haskell-src-exts.
  - Queries on ASTs: syb, uniplate.
- Comparison with
  - A small and (supposedly) high-quality Haskell project.
    - XMonad: a tiling window manager for X.
  - A big and (supposedly) hard to maintain Haskell project.
    - GHC: a state-of-the-art, open source, compiler and interactive environment for the functional language Haskell.

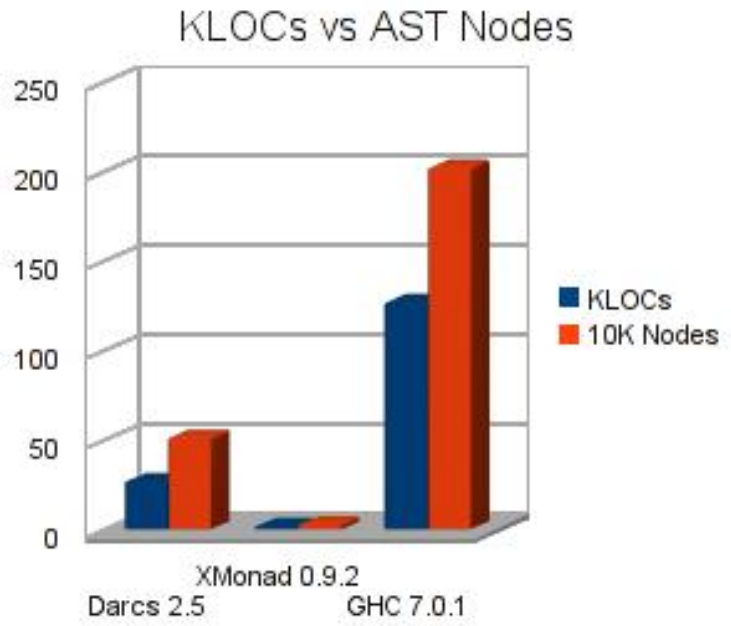
# Volume

| | |
|---|---|
| Darcs 2.5 | ★★★★★ |
| XMonad 0.9.2 | ★★★★★ |
| GHC 7.0.1 | ★★★★ |




- Does some big Haskell project exist?
- 656 Haskell KLOCs?

# KLOCs vs AST Nodes




- 10K Nodes $\approx 2\times$ KLOC ?
- 5 Nodes $\approx$ 1 LOC ?

# CC of functional code?

```
cleverCommute :: CommuteFunction -> CommuteFunction
cleverCommute c (p1:<p2) =
    case c (p1 :< p2) of
    Succeeded x -> Succeeded x
    Failed -> Failed
    Unknown -> case c (invert p2 :< invert p1) of
               Succeeded (p1' :< p2') -> Succeeded (invert p2' :< invert p1')
               Failed -> Failed
               Unknown -> Unknown
```

# CC of functional code?

```
cleverCommute :: CommuteFunction -> CommuteFunction
cleverCommute c (p1:<p2) =
    case c (p1 :< p2) of
    Succeeded x -> Succeeded x
    Failed -> Failed
    Unknown -> case c (invert p2 :< invert p1) of
               Succeeded (p1' :< p2') -> Succeeded (invert p2' :< invert p1')
               Failed -> Failed
               Unknown -> Unknown
```

**Cyclomatic complexity of 5**

# CC of functional code?

```
removeSubsequenceRL :: (MyEq p, Commute p) => RL p C(ab abc)
                       -> RL p C(a abc) -> Maybe (RL p C(a ab))
removeSubsequenceRL a b | lengthRL a > lengthRL b = Nothing
                        | otherwise = rsRL a b
    where rsRL :: (MyEq p, Commute p) => RL p C(ab abc)
                                         -> RL p C(a abc)
                                         -> Maybe (RL p C(a ab))
          rsRL NilRL ys = Just ys
          rsRL (x:<:xs) yys = removeRL x yys >>= removeSubsequenceRL xs
```

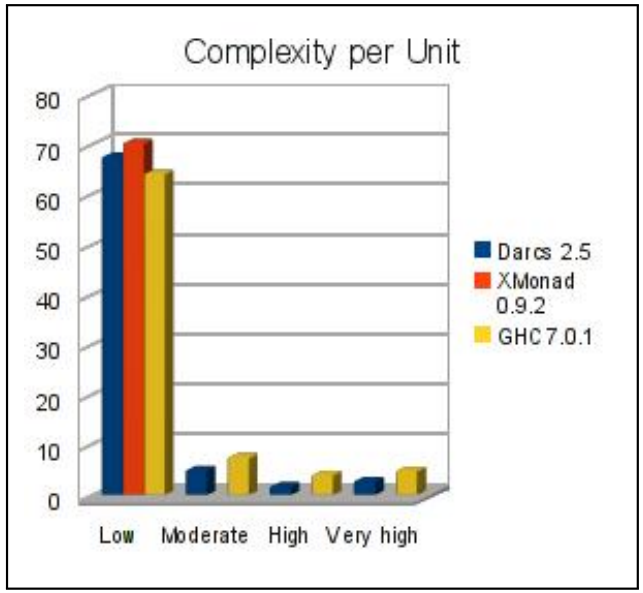
# CC of functional code?

```
removeSubsequenceRL :: (MyEq p, Commute p) => RL p C(ab abc)
                       -> RL p C(a abc) -> Maybe (RL p C(a ab))
removeSubsequenceRL a b | lengthRL a > lengthRL b = Nothing
                        | otherwise = rsRL a b
    where rsRL :: (MyEq p, Commute p) => RL p C(ab abc)
                                         -> RL p C(a abc)
                                         -> Maybe (RL p C(a ab))
          rsRL NilRL ys = Just ys
          rsRL (x:<:xs) yys = removeRL x yys >>= removeSubsequenceRL xs
```

**Cyclomatic complexity of 4**

# Complexity per unit

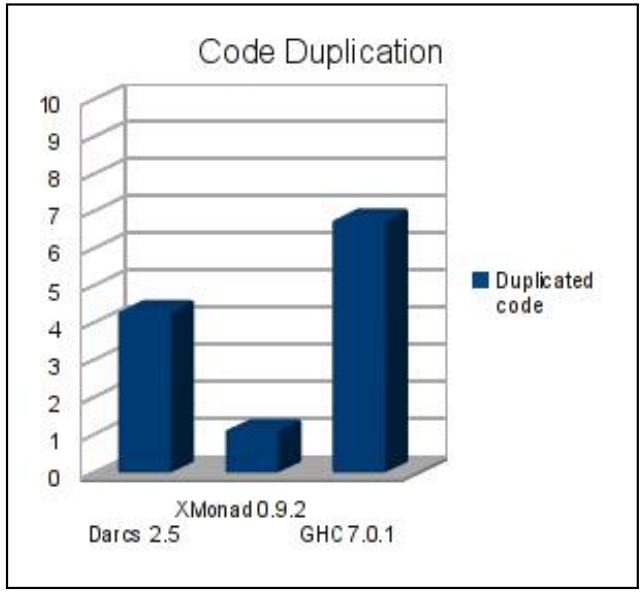
| | |
|---|---|
| Darcs 2.5 | ★★ |
| XMonad 0.9.2 | ★★★★★ |
| GHC 7.0.1 | ★ |




- An approximation to CC.
- Darcs: Sometimes abuse of complex local definitions, sometimes no refactoring effort.
- XMonad: Simply simple.
- GHC: Lot of data types with lot of data constructors.

# Code duplication

| | |
|---|---|
| Darcs 2.5 | ★★★★ |
| XMonad 0.9.2 | ★★★★★ |
| GHC 7.0.1 | ★★★ |




- Duplicated blocks of at least 4 lines.
- Darcs: A bad choice implied one star less.
- XMonad: All duplicated code is in tests.
- GHC: Most (Haskell) duplicated code in code generation.
  - Intermediate code generation has 25% of duplicated code.
  - Native code generation has 18% of duplicated code.
  - For C code RTS parallel support is an important source of duplication.

# Unit Size $\in (0, 300]$: Low Risk

```
nontrivialTriple :: RealPatch :> RealPatch :> RealPatch -> Bool
nontrivialTriple (a :> b :> c) =
    case commute (a :> b) of
    Nothing -> False
    Just (b' :> a') ->
      case commute (a' :> c) of
      Nothing -> False
      Just (c'' :> a'') ->
        case commute (b :> c) of
        Nothing -> False
        Just (c' :> b'') -> (not (a `unsafeCompare` a') || not (b `unsafeCompare` b')) &&
                            (not (c' `unsafeCompare` c) || not (b'' `unsafeCompare` b)) &&
                            (not (c'' `unsafeCompare` c) || not (a'' `unsafeCompare` a'))
```

# Unit Size $\in (300, 600]$: Moderate Risk

```
checkKnownShifts (ca, cb, sa, sb, ca', cb') = runST (
    do ca_arr <- newListArray (0, length ca) $ toBool (0:ca)
       cb_arr <- newListArray (0, length cb) $ toBool (0:cb)
       let p_a = listArray (0, length sa) $ B.empty:(toPS sa)
           p_b = listArray (0, length sb) $ B.empty:(toPS sb)
       shiftBoundaries ca_arr cb_arr p_a 1 1
       shiftBoundaries cb_arr ca_arr p_b 1 1
       ca_res <- fmap (fromBool . tail) $ getElems ca_arr
       cb_res <- fmap (fromBool . tail) $ getElems cb_arr
       return $ if ca_res  == ca' && cb_res == cb' then []
                else ["shiftBoundaries failed on "++sa++" and "++sb++" with "
                      ++(show (ca,cb))++" expected "++(show (ca', cb'))
                      ++" got "++(show (ca_res, cb_res))++"\n"])
 where toPS = map (\c -> if c == ' ' then B.empty else BC.pack [c])
       toBool = map (>0)
       fromBool = map (\b -> if b then 1 else 0)
```

# Unit Size $\in (600, 1100]$: High Risk

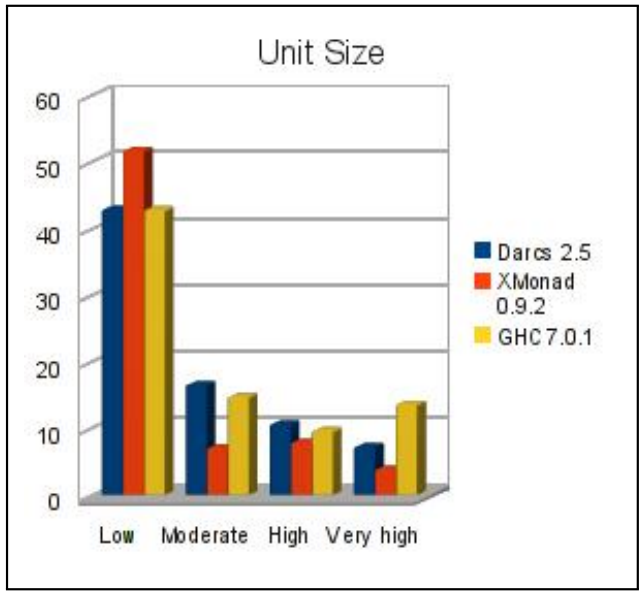
```
encode ps _ _ bufi | B.null ps = return bufi
encode ps n buf bufi = case B.head ps of
  c | c == newline ->
        do poke (buf `plusPtr` bufi) newline
           encode ps' qlineMax buf (bufi+1)
    | n == 0 && B.length ps > 1 ->
        do poke (buf `plusPtr` bufi) equals
           poke (buf `plusPtr` (bufi+1)) newline
           encode ps qlineMax buf (bufi + 2)
    | (c == tab || c == space) ->
        if B.null ps' || B.head ps' == newline
        then do poke (buf `plusPtr` bufi) c
                poke (buf `plusPtr` (bufi+1)) equals
                poke (buf `plusPtr` (bufi+2)) newline
                encode ps' qlineMax buf (bufi + 3)
        else do poke (buf `plusPtr` bufi) c
                encode ps' (n - 1) buf (bufi + 1)
    | (c >= bang && c /= equals && c <= tilde) ->
        do poke (buf `plusPtr` bufi) c
           encode ps' (n - 1) buf (bufi + 1)
    | n < 3 ->
        encode ps 0 buf bufi
    | otherwise ->
        do let (x, y) = c `divMod` 16
               h1 = intToUDigit x
               h2 = intToUDigit y
           poke (buf `plusPtr` bufi) equals
           poke (buf `plusPtr` (bufi+1)) h1
           poke (buf `plusPtr` (bufi+2)) h2
           encode ps' (n - 3) buf (bufi + 3)
    where ps' = B.tail ps
          newline = B.c2w '\n'
          tab     = B.c2w '\t'
          space   = B.c2w ' '
          bang    = B.c2w '!'
          tilde   = B.c2w '~'
          equals  = B.c2w '='
          intToUDigit i
            | i >= 0  && i <= 9  = B.c2w '0' + i
            | i >= 10 && i <= 15 = B.c2w 'A' + i - 10
            | otherwise = error $ "intToUDigit: '"++show i++"'not a digit"
```

# Unit Size > 1100: Very High Risk

```
urlThread ch = do junk <- flip showHex "" `fmap` randomRIO rrange
                  evalStateT urlThread' (UrlState Map.empty emptyQ 0 junk)
    where rrange = (0, 2^(128 :: Integer) :: Integer)
          urlThread' = do
            empty <- liftIO $ isEmptyChan ch
            st <- get
            let l = pipeLength st
                v = waitToStart st
            reqs <- if not empty || (nullQ v && l == 0)
                    then liftIO readAllRequests
                    else return []
            mapM_ addReq reqs
            checkWaitToStart
            waitNextUrl
            urlThread'
          readAllRequests = do
            r <- readChan ch
            debugMessage $ "URL.urlThread ("++url r++"\n"++
                           "          -> "++file r++")"
            empty <- isEmptyChan ch
            reqs <- if not empty
                    then readAllRequests
                    else return []
            return (r:reqs)
          addReq r = do
            let u = url r
                f = file r
                c = cachable r
            d <- liftIO $ alreadyDownloaded u
            if d
               then dbg "Ignoring UrlRequest of URL that is already downloaded."
               else do
                 st <- get
                 let p = inProgress st
                     v = waitToStart st
                     e = (f, [], c)
                     new_v = case priority r of
                               High -> pushQ u v
                               Low  -> insertQ u v
                     new_st = st { inProgress = Map.insert u e p
                                 , waitToStart = new_v }
                 case Map.lookup u p of
                   Just (f', fs', c') -> do
                     let new_c = minCachable c c'
                     when (c' /= c') $ let new_p = Map.insert u (f', fs', new_c) p
                                       in do modify (\s -> s { inProgress = new_p })
                                             dbg $ "Changing "++url++" request cachability from "++show c++" to "++show new_c
                     when (u `elemQ` v && priority r == High) $ do
                       modify (\s -> s { waitToStart = pushQ u (deleteQ u v) })
                       dbg $ "Moving "++url++" to head of download queue."
                     if f `notElem` (f':fs')
                       then let new_p = Map.insert u (f', f:fs', new_c) p
                            in do modify (\s -> s { inProgress = new_p })
                                  dbg "Adding new file to existing UrlRequest."
                       else dbg "Ignoring UrlRequest of file that's already queued."
                   _ -> put new_st
          alreadyDownloaded u = do
            n <- liftIO $ withMVar urlNotifications (return . (Map.lookup u))
            case n of
              Just v  -> not `fmap` isEmptyMVar v
              Nothing -> return True
```

# Unit size

| | |
|---|---|
| Darcs 2.5 | ★ |
| XMonad 0.9.2 | ★★ |
| GHC 7.0.1 | ★ |




- Darcs: Sometimes abuse of local definitions, sometimes no refactoring effort.
- XMonad: Little refactoring effort and it would receive four stars...
- GHC: Sometimes pattern matching against large data types, sometimes no refactoring effort.

# Module coupling

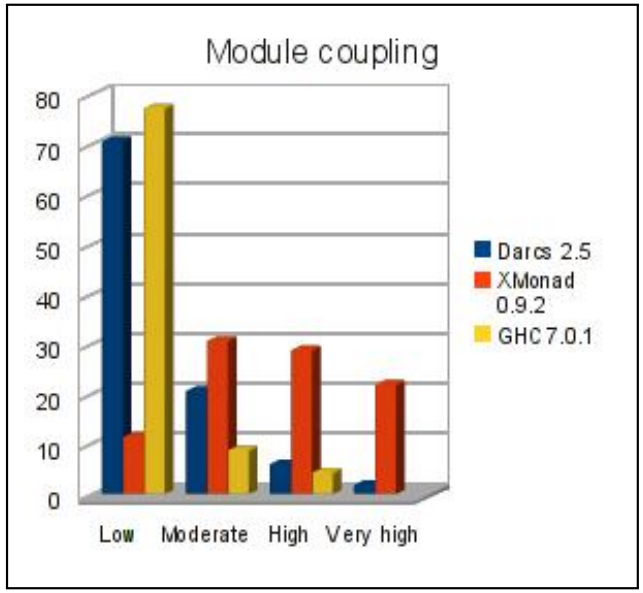
- How many code is affected if module *M* is modified?
- Volume of code depending on a given module:

| Dependent Code | Risk |
|---|---|
| 0%-10% | Low |
| 10%-30% | Moderate |
| 30%-60% | High |
| 60%-100% | Very high |

- Module coupling (mainly) influences *changeability* and *stability*.

# Module coupling

| | |
|---|---|
| Darcs 2.5 | ★★ |
| XMonad 0.9.2 | ★ |
| GHC 7.0.1 | ★★★★ |




- Rates were reversed... GHC is the best, XMonad is the worst.
- GHC modules are (40%/70%) bigger than Darcs/XMonad ones.
- Correlated with volume?
- Is GHC more stable than Darcs and XMonad?

# Unit testing

Only for Darcs 2.5 (no comparison needed)

- ★★ rate.
- 29% of top-level declarations, 28% of expressions.
- System tests cover about 75% of code.
- `Darcs.Patch` coverage

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| module Darcs.Patch | - | 0/0 | | - | 0/0 | | - | 0/0 | |
| module Darcs.Patch.Apply | 0% | 0/27 | | 0% | 0/158 | | 0% | 0/1144 | |
| module Darcs.Patch.Braced | 50% | 2/4 | | 50% | 2/4 | | 45% | 9/20 | |
| module Darcs.Patch.Commute | 75% | 6/8 | | 100% | 14/14 | | 83% | 103/123 | |
| module Darcs.Patch.ConflictMarking | 0% | 0/10 | | 0% | 0/26 | | 0% | 0/241 | |
| module Darcs.Patch.FileName | 55% | 11/20 | | 41% | 16/39 | | 44% | 110/249 | |
| module Darcs.Patch.Format | 100% | 1/1 | | - | 0/0 | | 100% | 1/1 | |
| module Darcs.Patch.Info | 23% | 11/46 | | 9% | 6/61 | | 24% | 166/670 | |
| module Darcs.Patch.Inspect | 0% | 0/4 | | - | 0/0 | | 0% | 0/19 | |
| module Darcs.Patch.Invert | 66% | 2/3 | | 100% | 4/4 | | 80% | 12/15 | |
| module Darcs.Patch.MatchData | 0% | 0/4 | | - | 0/0 | | 0% | 0/6 | |
| module Darcs.Patch.Merge | 66% | 2/3 | | 80% | 4/5 | | 76% | 48/63 | |
| module Darcs.Patch.Named | 17% | 5/28 | | 25% | 1/4 | | 27% | 47/172 | |
| module Darcs.Patch.OldDate | 0% | 0/30 | | 0% | 0/37 | | 0% | 0/922 | |
| module Darcs.Patch.Patchy | - | 0/0 | | - | 0/0 | | - | 0/0 | |
| module Darcs.Patch.Patchy.Instances | - | 0/0 | | - | 0/0 | | - | 0/0 | |
| module Darcs.Patch.Permutations | 51% | 16/31 | | 49% | 35/71 | | 50% | 264/518 | |
| module Darcs.Patch.Prim | 54% | 79/146 | | 51% | 160/311 | | 59% | 1426/2401 | |
| module Darcs.Patch.Read | 93% | 31/33 | | 68% | 11/16 | | 86% | 350/404 | |
| module Darcs.Patch.ReadMonads | 85% | 36/42 | | 60% | 23/38 | | 79% | 283/355 | |
| module Darcs.Patch.RegChars | 100% | 5/5 | | 40% | 6/15 | | 42% | 37/87 | |
| module Darcs.Patch.RepoPatch | - | 0/0 | | - | 0/0 | | - | 0/0 | |
| module Darcs.Patch.Show | 0% | 0/9 | | - | 0/0 | | 0% | 0/43 | |
| module Darcs.Patch.V1 | - | 0/0 | | - | 0/0 | | - | 0/0 | |
| module Darcs.Patch.V1.Apply | 0% | 0/3 | | 0% | 0/4 | | 0% | 0/26 | |
| module Darcs.Patch.V1.Commute | 59% | 26/44 | | 61% | 67/109 | | 55% | 526/945 | |
| module Darcs.Patch.V1.Core | 80% | 4/5 | | 57% | 4/7 | | 33% | 6/18 | |
| module Darcs.Patch.V1.Read | 100% | 4/4 | | 75% | 3/4 | | 92% | 51/55 | |
| module Darcs.Patch.V1.Show | 40% | 2/5 | | 66% | 2/3 | | 68% | 24/35 | |
| module Darcs.Patch.V1.Viewing | 25% | 1/4 | | 0% | 0/2 | | 9% | 1/11 | |
| module Darcs.Patch.V2 | - | 0/0 | | - | 0/0 | | - | 0/0 | |
| module Darcs.Patch.V2.Non | 83% | 20/24 | | 70% | 22/31 | | 73% | 216/292 | |
| module Darcs.Patch.V2.Real | 67% | 43/64 | | 58% | 130/221 | | 68% | 1331/1933 | |
| module Darcs.Patch.Viewing | 6% | 3/44 | | 3% | 4/106 | | 2% | 25/871 | |

# Maintainability

| | Analysability | Changeability | Stability | Testability | Maintainability |
|---|---|---|---|---|---|
| Darcs 2.5 | ★★★ | ★★★ | ★★ | ★★ | ★★ |
| XMonad 0.9.2 | ★★★★* | ★★★★ | ★* | ★★★★* | ★★★* |
| GHC 7.0.1 | ★★★* | ★★★ | ★★★★* | ★* | ★★★* |

# Testing Darcs' Patch Theory Kernel

8. Current state of Darcs unit tests

9. Improving coverage of existing QuickCheck generators
   - Reducing generation of empty trees
   - Rejecting useless test cases

10. Re-design and development of primitive patches testing
    - A new repository model
    - Generation of Primitive Patches
    - Coverage analysis
    - Summary

11. Conclusions and future work

# Background: QuickCheck


- A tool for testing Haskell programs automatically.
- The programmer provides properties which functions should satisfy.

```
prop_take n xs = take n xs `isPrefixOf` xs
```

- QuickCheck tests that the properties hold in a number of randomly generated cases.

```
+++ OK, passed 100 tests
```

- QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators.

```
vectorOf 5 (choose (1,10))
```

# Existing test code

`Darcs.Test.Patch.*` modules

- `Check`: Some kind of repository model.
- `Examples`: Check interesting properties on a set of pre-defined patches.
- **Examples2**: Set of interesting patches to test.
- `Info`: Generators and properties for patches metadata (encoding/decoding).

# Existing test code

`Darcs.Test.Patch.*` modules

- **Properties**: Interesting properties about patches.
- `Properties2`: Interesting properties about V1 patches using `Test` generators.
- **QuickCheck**: Generators for Prim and V2 patches (patches are valid by construction).
- `Test`: Generators for Prim and V1 (filter valid patches) based on `Check` module.
- `Unit`: HUnit test suite.
- **Unit2**: QuickCheck test suite.
- **Utils**: A few utilities.

# Coverage of V2 patches

```
data RealPatch prim C(x y) where
    Duplicate :: Non (RealPatch prim) C(x) -> RealPatch prim C(x x)
    Etacilpud :: Non (RealPatch prim) C(x) -> RealPatch prim C(x x)
    Normal :: prim C(x y) -> RealPatch prim C(x y)
    Conflictor :: [Non (RealPatch prim) C(x)] -> FL prim C(x y)
                  -> Non (RealPatch prim) C(x) -> RealPatch prim C(y x)
    InvConflictor :: [Non (RealPatch prim) C(x)] -> FL prim C(x y)
                     -> Non (RealPatch prim) C(x) -> RealPatch prim C(x y)
```

In short:

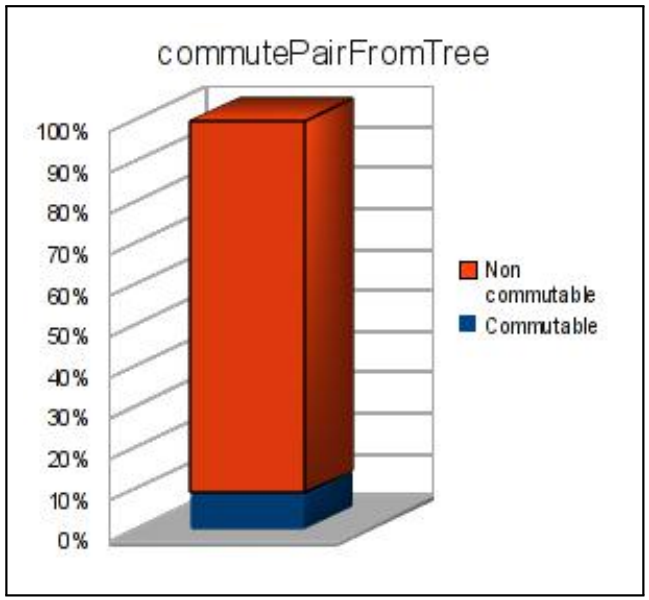
- Changes are represented by primitive patches.
- In case of conflict a special *conflictor* patch is used to represent the conflict.
  - Merge always "succeeds", but may produce conflicts.

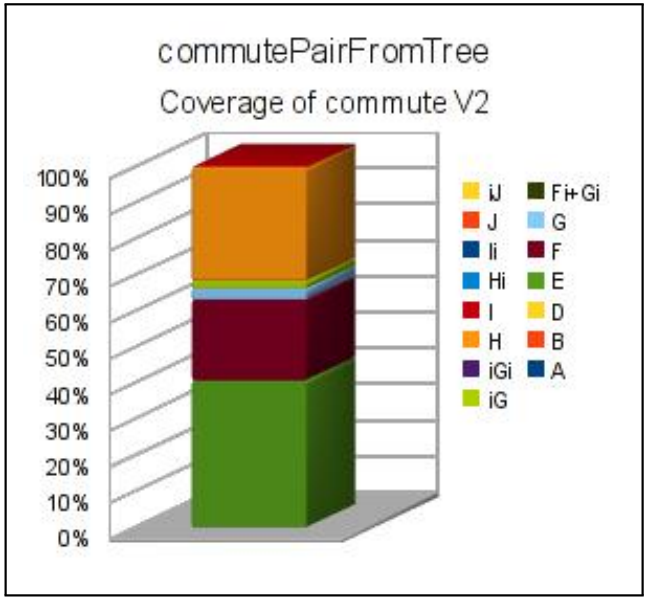
# Coverage of V2 patches

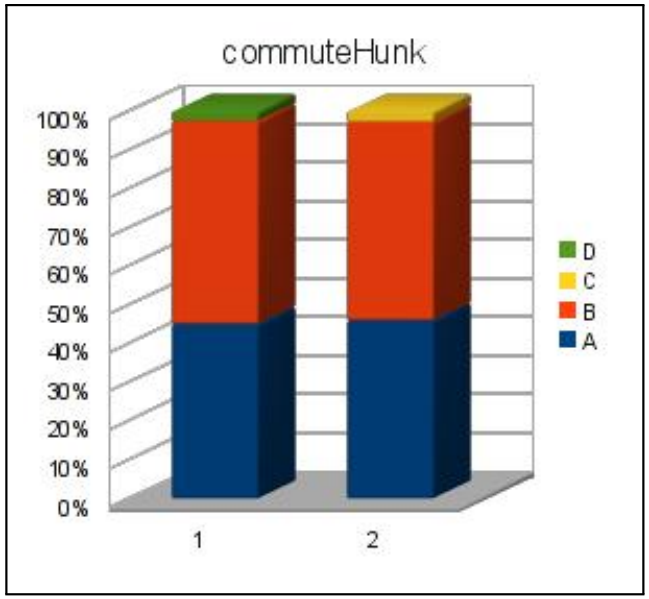
Generation of V2 patch pairs (aims to produce conflicts)

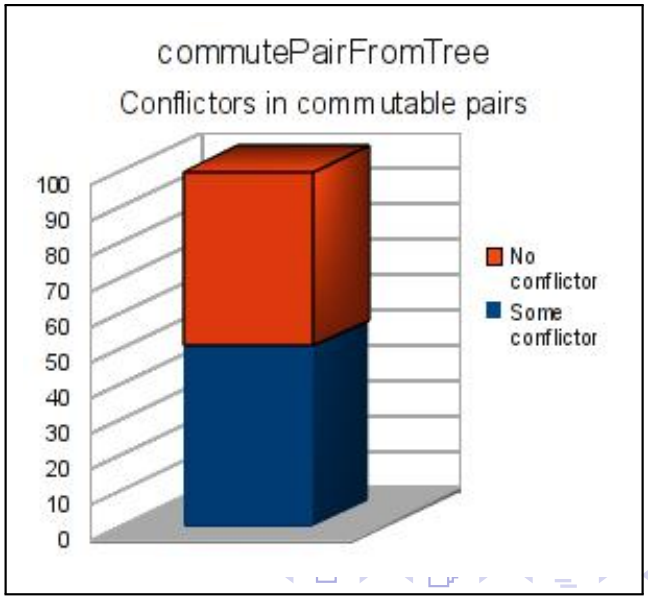
1. Generate a tree of hunk patches.
   - Simulating branches.
2. Flatten the tree using merge.
3. Take the last pair of patches.

# Coverage of V2 patches

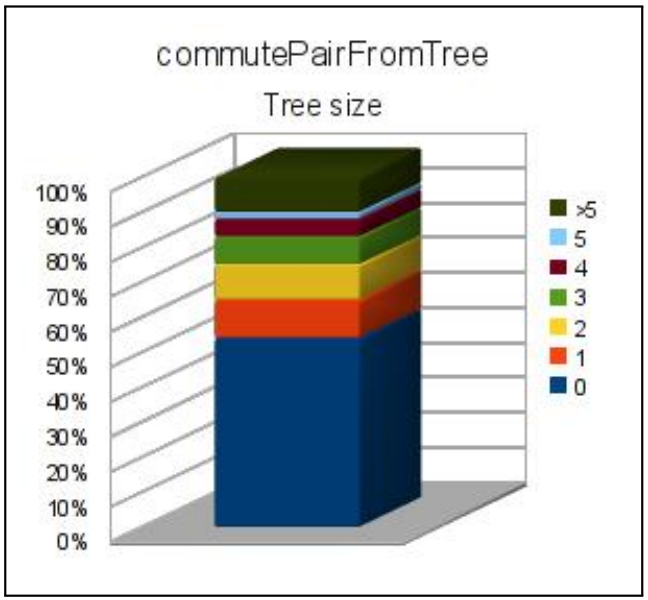


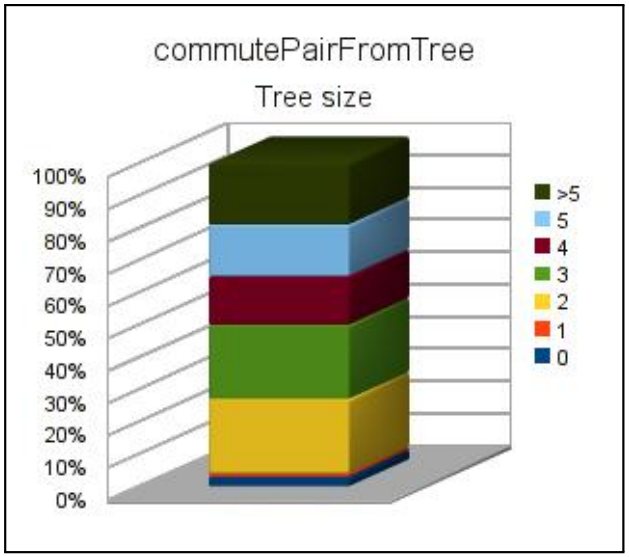



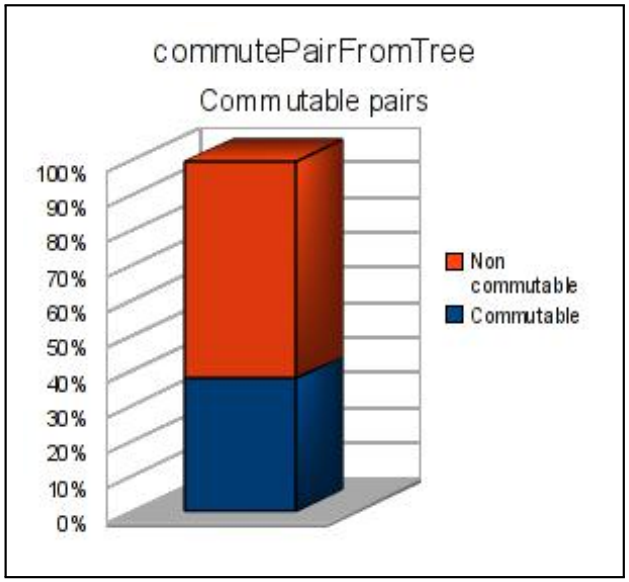
# Excessive number of empty trees

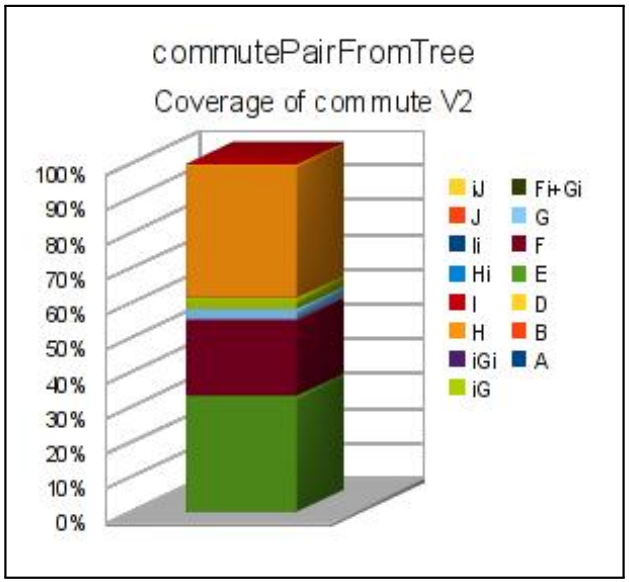



- Trees with size $\leq 1$ cannot produce any patch pair.
- In these cases `commutePairFromTree` use a default patch pair.
  - Useless for testing purposes.
  - Non commutable.

# Reducing generation of empty trees



# Useless test cases problem

- Most interesting properties are of the form $\forall P, Q, R, S : PQ \leftrightarrow RS : \ldots$
- Darcs properties result type is `Maybe Doc`

```
<property> :: Patchy p => (p :> p) -> Maybe Doc
<property> = case commute (x :> y) of
                  Nothing         -> Nothing     -- Useless
                  Just (y' :> x') ->
                        ...
                        case <Some Expression> of
                              <Failed>    -> Just <Error Message>
                              <Succeeded> -> Nothing
```

- Generators produce low rate of commutable pairs.
- Properties must be testable with any testing tool: QuickCheck, HUnit, ...
  - Prevents use of QuickCheck ==> operator.

# Testable TestResult

```
data TestResult = TestSucceeded
                | TestFailed Doc
                | TestRejected

succeeded :: TestResult
failed :: Doc     -- ^ Error message
          -> TestResult
-- | Rejects test case
rejected :: TestResult

...

isFailed :: TestResult -> Bool
-- | A test is considered OK if it does not fail.
isOk :: TestResult -> Bool

instance Testable TestResult where
  property = ...
```

# Current strategy

## How `Darcs.Test.Patch.QuickCheck` generate primitive patches ?

```
data RepoModel
 = RepoModel {
     rmFileName :: !FileName,
     rmFileContents :: [B.ByteString]
   } deriving (Eq)

arbitraryFP :: RepoModel -> Gen (Prim, RepoModel)

arbitraryHunk :: [B.ByteString] -> Gen (FilePatchType, [B.ByteString])
```

Strengths:

- Test cases are valid by construction.
- It is possible to reproduce a test case on disk.

Weaknesses:

- Only hunks are covered.
- Low rate of commutable pairs.
- Needs custom code for patch application.
  - No way to test *apply* code.

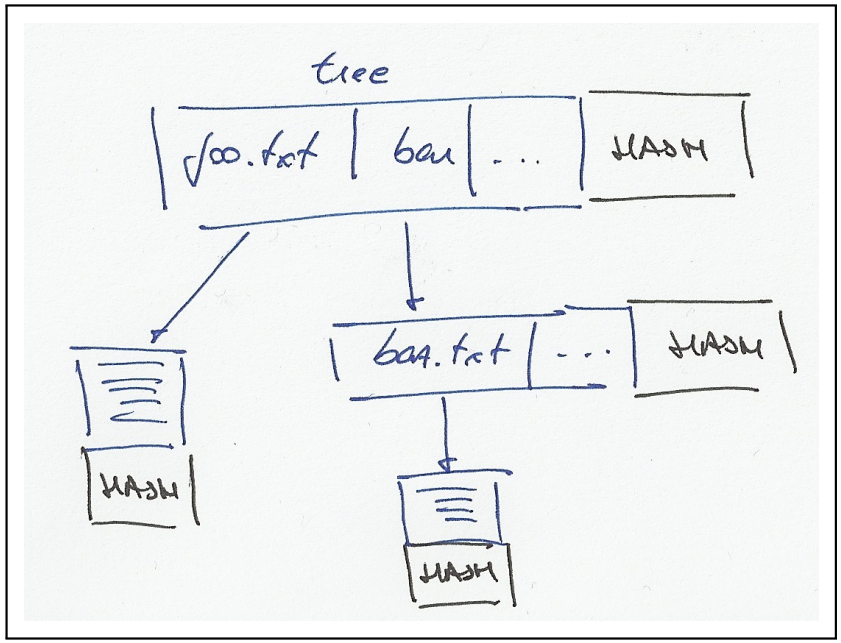
# Hashed Storage

*"Support code for reading and manipulating hashed file storage (where each file and directory is associated with a cryptographic hash, for corruption-resistant storage and fast comparisons)."*

- `Storage.Hashed.Tree`: *"The abstract representation of a Tree and useful abstract utilities to handle those."*

```
data Blob m = Blob !(m BL.ByteString) !Hash
data TreeItem m = File !(Blob m)
                | SubTree !(Tree m)
                | Stub !(m (Tree m)) !Hash

data Tree m = Tree { items :: (M.Map Name (TreeItem m))
                   , treeHash :: !Hash }
```




- Darcs repositories are handled through Hashed Storage.
  - It is possible to apply a patch to a Hashed Storage `Tree`!

# Repository model

New module `Darcs.Test.Patch.RepoModel`:

- A repository model is a wrapper over a Hashed Storage `Tree`.

```
newtype RepoModel = RepoModel { repoTree :: Tree Maybe }

newtype RepoItem = RepoItem { treeItem :: TreeItem Maybe }

type Content = [B.ByteString]
type File = RepoItem
type Dir  = RepoItem
```

- It offers a simplified and more specific API.
- It is possible to compare trees;

```
instance Eq RepoModel where
  repo1 == repo2 = ...
```

- and apply patches to them.

```
applyPatch :: Apply patch => patch -> RepoModel -> Maybe RepoModel
applyPatch patch (RepoModel tree) = RepoModel <$> applyToTree patch tree
```

# Generation of repositories

$aRepo :: Int \rightarrow Int \rightarrow Gen\ RepoModel$

$aRepo\ files\#_{max}\ dirs\#_{max}$

1. Arbitrarily choose $files\# \in [0, files\#_{max}]$.
2. Arbitrarily choose $subdirs\# \in [0, dirs\#_{max}]$.
3. $filesPerDir\# := \frac{files\#_{max} - files\#}{subdirs\#}$.
4. $subdirsPerDir\# := \frac{dirs\#_{max} - subdirs\#}{subdirs\#}$.
5. Generate $files\#$ files.
6. Generate $subdirs\#$ directories with up to $filesPerDir\#$ files and up to $subdirsPerDir\#$ subdirectories.

# Generating primitive patches

Strategy:

- Generate a (small) repository.
- Generate a patch applicable to that repository.

Problem: patches have pre-conditions, it is only possible to generate a subset of patch types given a repository.

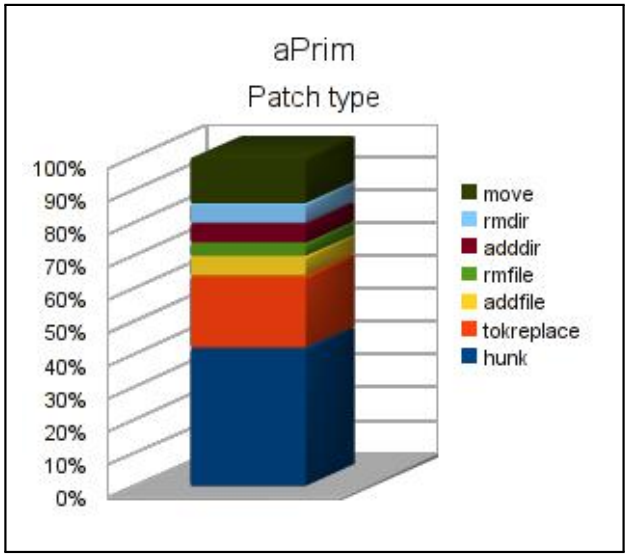
(a) Select a patch type arbitrarily, fail if pre-conditions are violated.
   - `Gen (Maybe Prim)`
   - Less robust, potentially inefficient.

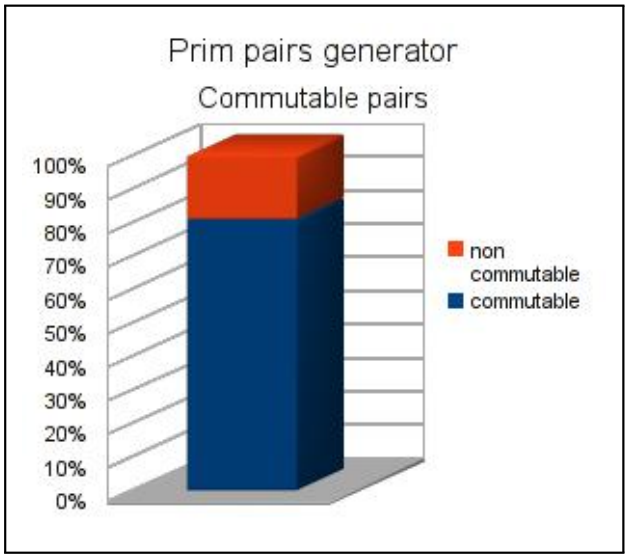
(b) Frequencies table for selecting patches, whose entries are conditionally enable.

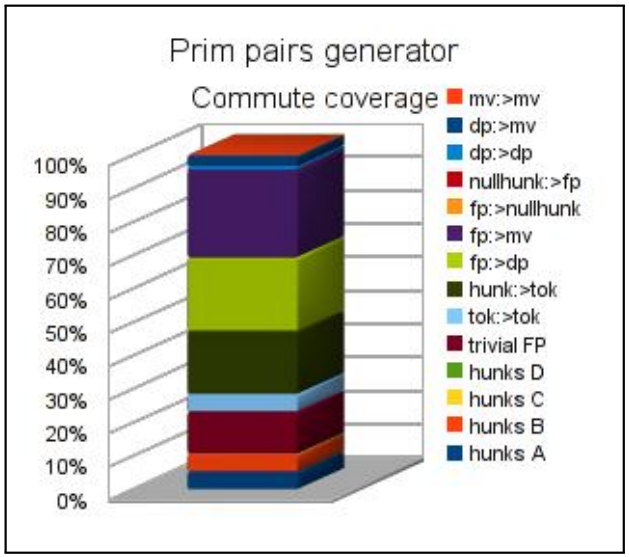
```
[ ( if isJust mbFile then 15 else 0
  , aHunkP $ fromJust mbFile )
, ... ]
```

   - More robust, efficient.

# Coverage analysis



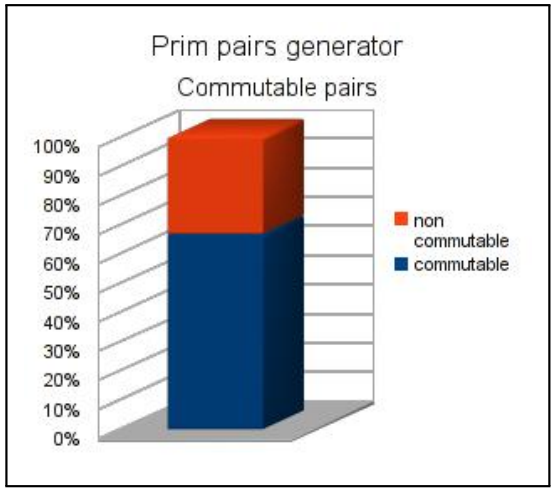
# Improving coverage for commutable hunks

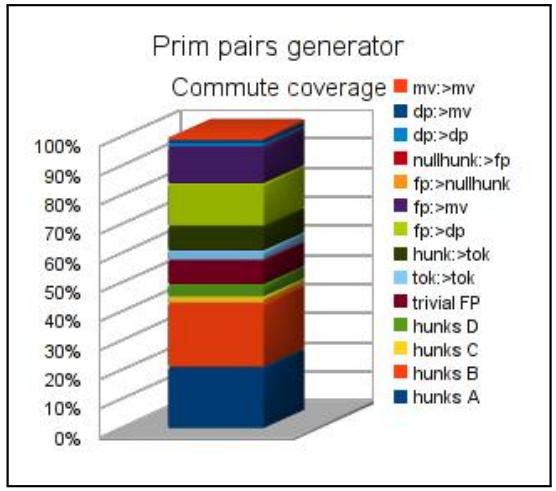
```
-- Try to generate commutable pairs of hunks
hunkPairP :: (AnchoredPath,File) -> Gen (Prim :> Prim)

aPrimPair :: RepoModel -> Gen (Prim :> Prim, RepoModel)
aPrimPair repo
  = do mbFile <- maybeOf repoFiles
       frequency
          [ ( if isJust mbFile then 1 else 0
            , ""use hunkPairP""
            )
          , ( 1
            , ""use the default generator for Prim pairs""
            )
          ]
```

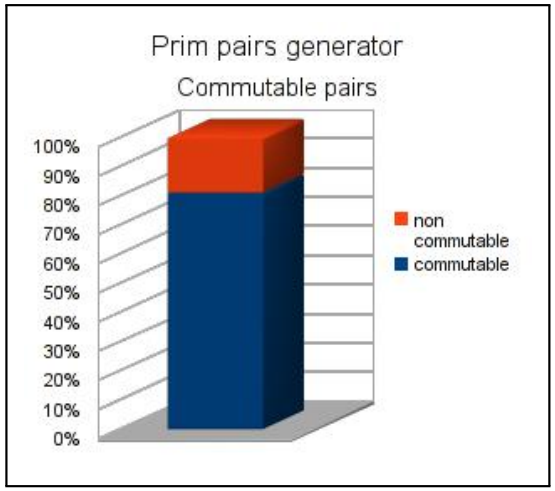
# Improving coverage for commutable hunks

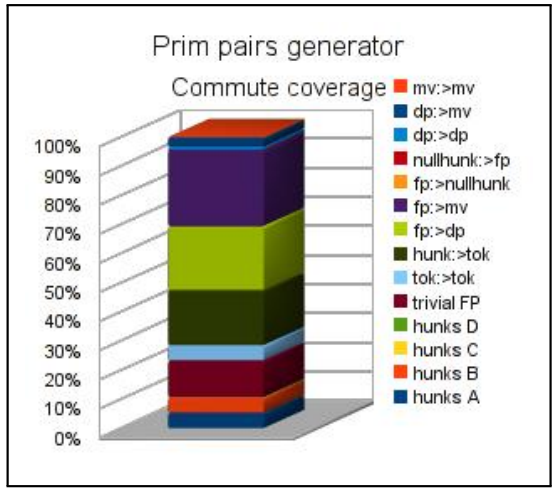
Now:




Before:



# Summary

- Properties like *invert rollback* or *effect preserving* are now being tested.
  - Now we know empty-hunks break *effect preserving* property.
- `Darcs.Patch.V1.Apply` is now almost fully tested.
- `Darcs.IO` Hashed Storage implementation is now being tested.
- `Darcs.Test.Patch.QuickCheck` and `Darcs.Test.Patch.Examples2` rewritten to make use of new repository model and patch generators.
  - Thanks to this we have found a possible bug in V2 *commute*/*merge* which breaks *commute* symmetry.
- Automatic generation of coverage report for both system and unit tests.

# Conclusions

- Darcs is not easy to maintain.
  - Metrics and experience agree.

A possible short-term plan:

- Avoid local declarations when they make sense as top-level.
  - Facilitates testing; reduces complexity and unit size.
- Avoid module "private utilities".
  - Use unit tests to ensure contracts are never broken.
- Write more unit tests (important Darcs weakness).

# Conclusions

- Patch logic is hard to test.
  - Some refactoring may help.
  - Write QuickCheck generators is tricky.
  - Small changes have a big impact in coverage.
  - Properties depend on conditions which are hard to fulfill.

# Future work

- Refine code metrics and write proper tools.
- Integrate code metrics into development process.
  - Run code metrics to guarantee code quality.
  - Just as you run tests to guarantee code correctness.
- Refactor, clean up and re-organize `Darcs.Test.Patch.*`.
- 100% coverage for `Darcs.Patch.*`.
- Extend/explore the usefulness of the new repository model.
  - More properties involving repository state.
  - Could we fully simulate Darcs in memory?

# Thanks to

- Ganesh Sittampalam.
- Petr Rockai.
- Jason Dagit.
- All FreeNode #darcs people.

# Questions?

Shoot!