Modeling Darcs' Theory of Patches with Alloy

Iago Abal   João Melo

November 28, 2010

Table of contents

Darcs Advanced Revision Control System

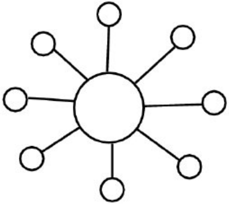
- A Revision Control System.
- Originally developed by the physicist David Roundy.
- Written in Haskell
  - First version written in C++ but..
    "C++ version was too buggy to be useful"
- It has several practical problems so it is mostly used by the Haskell community.

Key features:

- Distributed.
- Change-based.
- Strong "mathematical" background: Patch Theory.

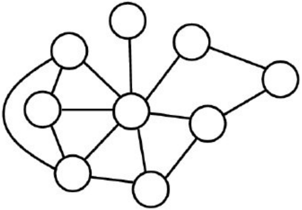
Distributed rather than centralized

Centralized



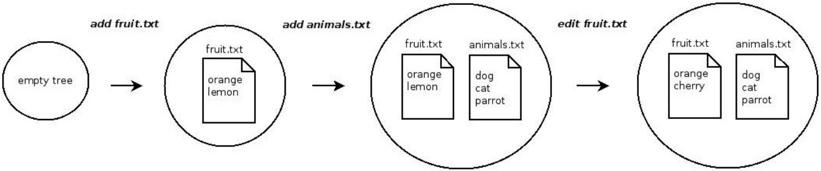
Examples: CVS, Subversion, Perforce

Distributed

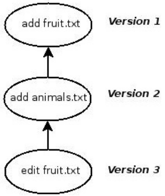


Examples: darcs, Git, Bitkeeper, monotone, arch

Change-based rather than version-based

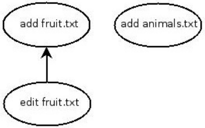


Version-based



Examples: Git, Bitkeeper, Monotone, CVS, Subversion

Change-based



Examples: darcs

Patch theory

- An algebra of patches.
- Developed by David Roundy.
- A number of patches types which define the possible modifications over a tree.
  - Add/remove a directory, add/remove/edit a file, ...
- Operations for apply, invert and commute patches.
- Properties that any implementation must ensure.
- Theorems that are supposed to hold.

Patch Theory soundness

> *I think a little background on the author is in order. I am a physicist, and think like a physicist.* **The proofs and theorems given here are what I would call "physicist" proofs and theorems**, *which is to say that while the proofs may not be rigorous, they are practical, and the theorems are intended to give physical insight.* **It would be great to have a mathematician work on this to give patch theory better formalized foundations.**

David Roundy

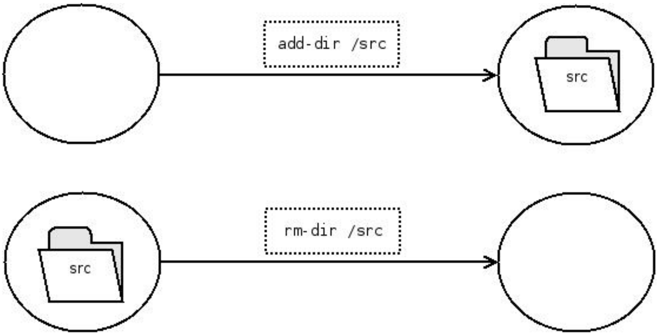
Darcs User Manual.

Patches

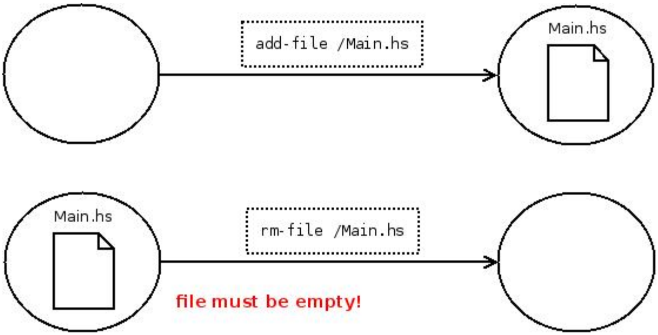
> *A patch describes a change to the tree. It could be either a primitive patch (such as a file add/remove, a directory rename, or a hunk replacement within a file), or a composite patch describing many such changes.*

David Roundy
Darcs User Manual.

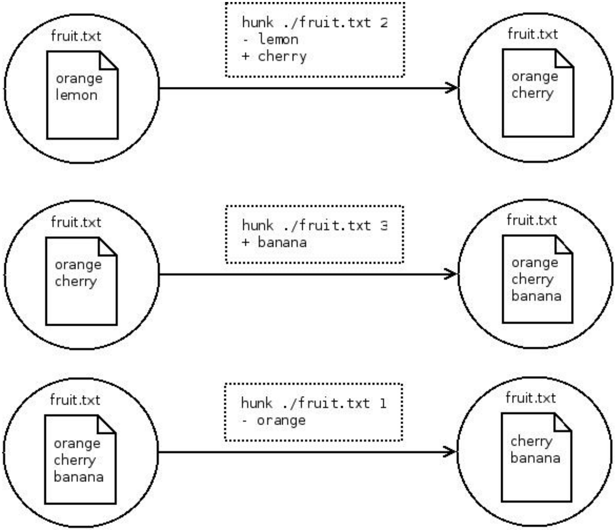
Directory-patches



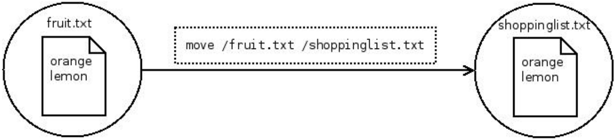
File-patches



File-patches



Move-patches



Patches

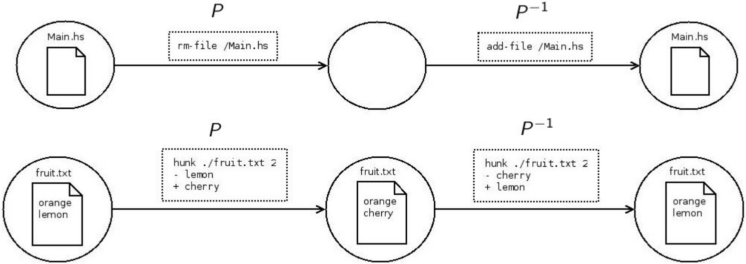
- A patch contains metadata describing a change over a tree.
- A patch could be view as an injective simple relation.
    - $p : \text{Tree} \to \text{Tree}$
    - This view is often called the *effect* of a patch.
    - Every patch can be applied to some tree.
      $\forall p : \text{Patch}, \mathit{domain}(p) \neq \emptyset$
- Patches may be composed into sequences.
    - $p_1 \cdot p_2 \cdot \ldots \cdot p_n$
    - A sequence of patches is sensible if its overall effect is not $\perp$.
        - p and q are said *sequential* iff $p \cdot q$ is sensible.
    - The history of a repository is a sequence of patches.

Inverse of a patch

> *The inverse of patch P is P^{-1}, which is the "simplest" patch for which the composition $P^{-1}P$ makes no changes to the tree.*

David Roundy
Darcs User Manual.

Inverse of a patch



Inverse of a patch

- The inverse of a patch undo the patch effect.
  Rollback $\forall t : \text{Tree}, p^{-1}(p(t)) = t$
  - The effect of the inverse is the converse of the effect.
- Invert : Patch $\rightarrow$ Patch is an injection.
  - **Every patch must be invertible.**
    - Otherwise you won't be able to rollback.
- **Theorem** $(p \cdot q)^{-1} = q^{-1} \cdot p^{-1}$

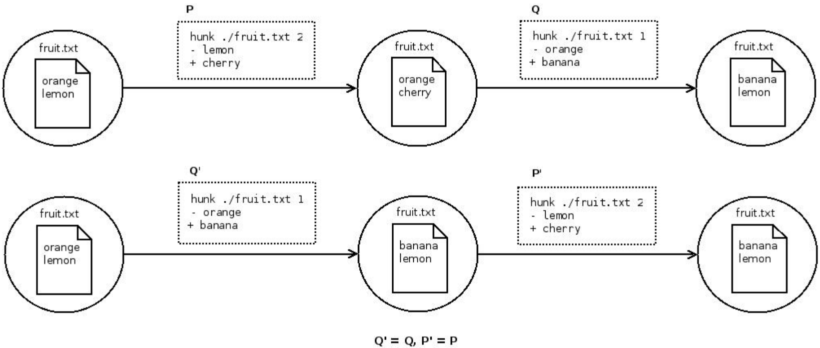
Patch commutation

> *Informally, a pair of pathes (p, q) commutes when we can find another pair (q', p'), with p' and q' having the same "meaning" as p and q respectively, such that $pq \equiv q'p'$.*

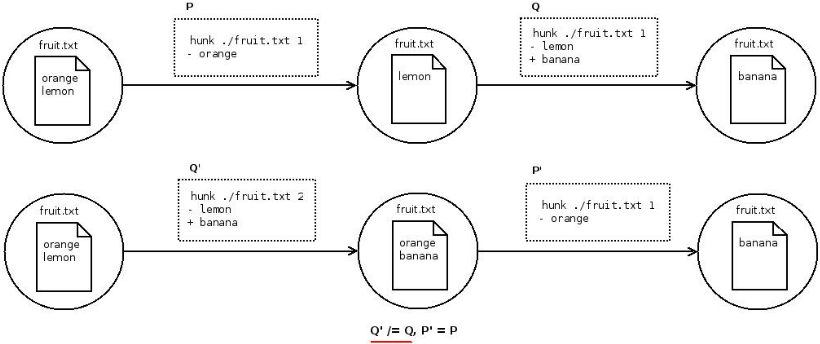
Judah Jacobson

A Formalization of Darcs Patch Theory using Inverse Semigroups.

Patch commutation



Patch commutation



Patch commutation

- $(p, q) \leftrightarrow (r, s)$
  - $\leftrightarrow$: Patch $\times$ Patch $\rightarrow$ Patch $\times$ Patch
  - Partial, defined only for p, q *sequential*.
  - Simple and injective.
  - Preserves *sequential*: r, s *sequential*.
  - **Symmetric**.
- **Effect preserving:** $(p, q) \leftrightarrow (r, s) \Rightarrow p \cdot q = r \cdot s$
- **Rotating:** $(p, q) \leftrightarrow (r, s) \Rightarrow (r^{-1}, p) \leftrightarrow (s, q^{-1})$

Overview

We model the core of Darcs core.

- Primitive patches only.
- We don't model sequences of patches (hard anyway).
- `Darcs.Patch.Prim`: `Prim` data type, invert and commute.
- `Darcs.Patch.Apply`: Application of patches.

Tree

```
class Apply patch where
    apply :: WriteableDirectory m => [DarcsFlag] -> patch -> m ()

class (Functor m, MonadPlus m) => ReadableDirectory m where
    mDoesDirectoryExist :: FileName -> m Bool
    mDoesFileExist :: FileName -> m Bool
    mReadFilePSs :: FileName -> m [B.ByteString]

class ReadableDirectory m => WriteableDirectory m where
    mWriteFilePSs :: FileName -> [B.ByteString] -> m ()
    mCreateDirectory :: FileName -> m ()
    mRemoveDirectory :: FileName -> m ()
    mCreateFile :: FileName -> m ()
    mRemoveFile :: FileName -> m ()
    mRename :: FileName -> FileName -> m ()
```

Tree

```
sig Path {
  parent : lone Path,
  name : Name
}

fun readFile[t : Tree, f : Path] : (seq Line)

pred CreateFile[t : Tree,  f : Path, t' : Tree]
pred RemoveFile[t : Tree, f : Path, t' : Tree]
pred CreateDir[t : Tree, d : Path, t' : Tree]
pred RemoveDir[t : Tree, d : Path, t' : Tree]
pred Rename[t : Tree, src : Path, dest : Path1, t' : Tree]
pred WriteFile[t : Tree,  f : Path, text : seq Line, t' : Tree]
```

Tree

- At the very first time we just model flat trees.
  - The "traditional" model of filesystem was used.

```
abstract sig FSObject {}
sig File extends FSObject {}
...
```

- Once you add directories this model is not good.
  - Equivalent filesystems are considered different.
  - You need to determine if an item is a child of another independently of any specific tree.

```
sig Path { parent : Dir, name : Name }
```

- Get the item pointed by a path (as a list of names) is no trivial without recursion.

Path-based Tree

```
sig Tree {
  Dirs   : set Path,
  Files : set Path,
  content : Path -> (seq Line)
}

pred Inv[t : Tree] {
  no (t.Dirs & t.Files)
  all x : t.Items | x.parent in t.Dirs
  t.content in t.Files -> (seq Line)
}
```

- Limitations when renaming items due to lack of recursion.

Patches types

```
data Prim where
    Move :: !FileName -> !FileName -> Prim
    DP :: !FileName -> !DirPatchType -> Prim
    FP :: !FileName -> !FilePatchType -> Prim

data FilePatchType = RmFile | AddFile
                   | Hunk !Int [B.ByteString] [B.ByteString]
                   deriving (Eq,Ord)

data DirPatchType = RmDir | AddDir
                    deriving (Eq,Ord)
```

Patches types

```
abstract sig Patch {}

abstract sig DirPatch extends Patch {
  path : Path
}
sig Adddir, Rmdir extends DirPatch {}

abstract sig FilePatch extends Patch {
  path : Path
}
sig Addfile, Rmfile extends FilePatch {}
sig Hunk extends FilePatch {
  line : Int,
  old : seq Line,
  new : seq Line
}

sig Move extends Patch {
  source : Path,
  dest : Path
}
```

Patch application

Recall:

```
class Apply patch where
    apply :: WriteableDirectory m => [DarcsFlag] -> patch -> m ()
```

- Concrete directory (tree) type.

```
pred Apply[t : Tree, p : Patch, t' : Tree] {
  ApplyDirpatch[t,p,t'] or ApplyFilepatch[t,p,t'] or ApplyMove[t,p,t']
}
```

- We need to define sequential for pre-conditions.

```
pred sequential[p, q : Patch] {
  some t1, t2, t3 : Tree |
      t1.Inv and Apply[t1,p,t2] and Apply[t2,q,t3]
}
```

Patch application

```
pred ApplyHunk[t : Tree, h : Hunk, t' : Tree] {
  -- PRE
  h in Hunk
  h.path in t.Files
  let text = t.readFile[h.path],
      old_next = h.line.add[#h.old], new_next = h.line.add[#h.new],
      old_end = old_next.prev, new_end = new_next.prev
  {
    old_end < #text and h.old = text.subseq[h.line,old_end] // old content is right
    pos[h.ldelta] and pos[#text] => text.lastIdx.add[h.ldelta] in seq/Int   // respect file size limit

    let text' = t'.readFile[h.path] {
      WriteFile[t,h.path,text',t']  // nothing but the content of the file pointed by h.path is changed

      -- CHANGE
      #text' = (#text).add[h.ldelta]
      text'.subseq[h.line,new_end] = h.new

      -- KEEP
      text'.subseq[0,h.line.prev] = text.subseq[0,h.line.prev]  // same preffix
      text'.subseq[new_next,text'.lastIdx] = text.subseq[old_next,text.lastIdx] // same rest
    }
  }
}
```

Patch inversion

```
class Invert patch where
    invert :: patch -> patch

instance Invert Prim where
    invert (FP f RmFile)  = FP f AddFile
    invert (FP f AddFile)  = FP f RmFile
    invert (FP f (Hunk line old new))  = FP f $ Hunk line new old
    invert (DP d RmDir) = DP d AddDir
    invert (DP d AddDir) = DP d RmDir
    invert (Move f f') = Move f' f
```

Patch inversion

```
pred Invert[p, p_inv : Patch] {
     InvertDirpatch[p, p_inv]
  or InvertFilepatch[p, p_inv]
  or InvertMove[p, p_inv]
}

pred InvertDirpatch[dp, dp_inv : DirPatch] {
  dp.InvertAdddir[dp_inv] or dp.InvertRmdir[dp_inv]
} ...

pred InvertFilepatch[fp, fp_inv : FilePatch] {
     fp.InvertAddfile[fp_inv]
  or fp.InvertRmfile[fp_inv]
  or fp.InvertHunk[fp_inv]
} ...

pred InvertMove[mv, mv_inv : Move] {
  mv_inv.source = mv.dest
  mv_inv.dest = mv.source
}
```

Universe is not saturated enough problem

```
assert EveryPatchIsInvertible {
  all p : Patch | p.Inv => some p_inv : Patch | p.Invert[p_inv]
}

check EveryPatchIsInvertible
```

- Alloy always finds a counterexample.
- Given a patch p there is no guarantee that p^{-1} will exist in a **finite** universe.
- Does a generator axiom make sense for this case?

Type headache (problem)

```
pred Invert[p, p_inv : Patch] {
     InvertHunk[p, p_inv] or InvertMove[p, p_inv]
}

pred InvertHunk[h, h_inv : Hunk] {
  ...
}

pred InvertMove[mv, mv_inv : Move] {
  mv_inv.source = mv.dest and mv_inv.dest = mv.source
}
```

- *h* is some hunk that adds some lines to a text file...
- `Invert[h,h]` $\rightarrow$?
  - `InvertHunk[h,h]` $\rightarrow$ **False**
  - `InvertMove[h,h]` $\stackrel{def}{=} \emptyset = \emptyset \wedge \emptyset = \emptyset$ $\rightarrow$ **True**
  - `Invert[h,h]` $\rightarrow$ **True !!!**

Type headache (solution)

```
pred Invert[p, p_inv : Patch] {
     InvertHunk[p, p_inv] or InvertMove[p, p_inv]
}

pred InvertHunk[h, h_inv : Hunk] {
  h in Hunk and h_inv in Hunk
  ...
}

pred InvertMove[mv, mv_inv : Move] {
  mv in Move and mv_inv in Move
  ...
}
```

- `Invert[h,h]` $\rightarrow$ **False**
    - $h \notin$ Move $\Rightarrow$ `InvertMove[h,h]` $\rightarrow$ **False**
- Why does not Alloy introduce these constraints?

Patch commutation

```
instance Commute Prim where
    commute x = toMaybe $ msum [speedyCommute x
                               ,cleverCommute commuteFiledir x
                               ]

speedyCommute :: CommuteFunction
speedyCommute (p1 :< p2) -- Deal with common case quickly!
    ...

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
```

Patch commutation

```
commuteFiledir :: CommuteFunction

commuteFiledir (FP f1 p1 :< FP f2 p2) =
  if f1 /= f2 then Succeeded (FP f2 p2 :< FP f1 p1)
  else commuteFP f1 (p1 :< p2)
commuteFiledir (DP d1 p1 :< DP d2 p2) = ...
commuteFiledir (DP d dp :< FP f fp) = ...

commuteFiledir (Move d d' :< FP f2 p2) = ...
commuteFiledir (Move d d' :< DP d2 p2) = ...
commuteFiledir (Move d d' :< Move f f') = ...

commuteFiledir _ = Unknown


commuteFP :: FileName -> (FilePatchType :< FilePatchType)
          -> Perhaps (Prim :< Prim)

commuteFP f (Hunk line2 old2 new2 :< Hunk line1 old1 new1) = seq f $
  toPerhaps $ commuteHunk f (Hunk line2 old2 new2 :< Hunk line1 old1 new1)
commuteFP _ _ = Unknown
```

Patch commutation

```
pred Commute[p, q, q', p' : Patch] {
     CommuteDirpatches[p,q,q',p']
  or CommuteFilepatches[p,q,q',p']
  or CommuteDirAndFilePatches[p,q,q',p']
  or CommuteMovePatches[p,q,q',p']
  or CommuteMoveAndDirFilePatches[p,q,q',p']
}

pred CommuteDirpatches[dp1, dp2, dp2', dp1' : DirPatch] { ... }

pred CommuteFilepatches[fp1, fp2, fp2', fp1' : FilePatch] {
  ...
  fp1.path != fp2.path => fp2' = fp2 and fp1' = fp1    // trivially commute
    else CommuteSFHunks[fp1,fp2,fp2',fp1']
}

pred CommuteDirAndFilePatches[p, q, q', p' : Patch] {
  CommuteFilepatchDirpatch[p,q,q',p'] or CommuteDirpatchFilepatch[p,q,q',p']
}

...
```

Patch commutation

```
commuteHunk :: FileName -> (FilePatchType :< FilePatchType)
            -> Maybe (Prim :< Prim)
commuteHunk f (Hunk line2 old2 new2 :< Hunk line1 old1 new1)
  | seq f $ line1 + #new1 < line2 =
      Just (FP f (Hunk line1 old1 new1) :<
            FP f (Hunk (line2 - #new1 + #old1) old2 new2))
  | line2 + #old2 < line1 =
      Just (FP f (Hunk (line1+ #new2 - #old2) old1 new1) :<
            FP f (Hunk line2 old2 new2))
  | line1 + #new1 == line2 &&
    #old2 /= 0 && #old1 /= 0 && #new2 /= 0 && #new1 /= 0 =
      Just (FP f (Hunk line1 old1 new1) :<
            FP f (Hunk (line2 - #new1 + #old1) old2 new2))
  | line2 + #old2 == line1 &&
    #old2 /= 0 && #old1 /= 0 && #new2 /= 0 && #new1 /= 0 =
      Just (FP f (Hunk (line1 + #new2 - #old2) old1 new1) :<
            FP f (Hunk line2 old2 new2))
  | otherwise = seq f Nothing
commuteHunk _ _ = impossible
```

Patch commutation

```
pred CommuteSFHunks[h1, h2, h2', h1' : Hunk] {
  -- PRE
  ...

  -- CHANGE
  h1.line.add[#h1.new] < h2.line
      => (h2'.line = h2.line.sub[h1.ldelta] and h1'.line = h1.line)
  else h2.line.add[#h2.old] < h1.line
      => (h2'.line = h2.line and h1'.line = h1.line.add[h2.ldelta])
  else (h1.line.add[#h1.new] = h2.line
        and h1.isReplaceHunk and h2.isReplaceHunk)
      => (h2'.line = h2.line.sub[h1.ldelta] and h1'.line = h1.line)
  else (h2.line.add[#h2.old] = h1.line
        and h1.isReplaceHunk and h2.isReplaceHunk
      and h2'.line = h2.line and h1'.line = h1.line.add[h2.ldelta])

  -- KEEP
  ...
}
```

int/Int overloading

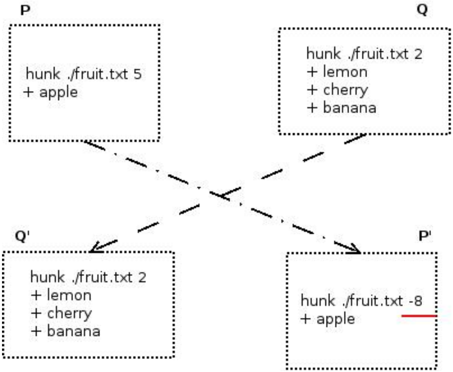
- Alloy automatically casts int to Int or vice-versa when needed.
- $+$ is addition for int, but set union for Int.
  - $1 + 1 = 2 \quad \text{VS} \quad \{1\} + \{1\} = \{1\}$
  - **Be careful!**

Integer overflow

Darcs uses Int data-type for line numbers.

`A fixed-precision integer type with at least the range` $[-2^{29}, 2^{29}-1]$

(Haskell98 Report)



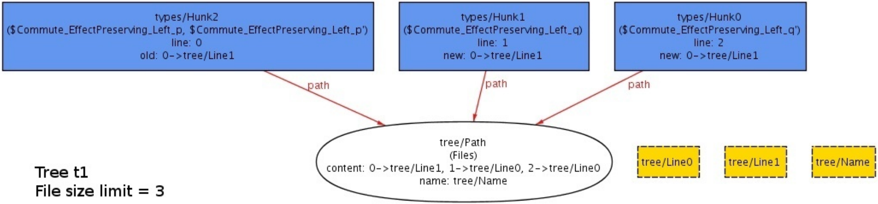
Going further with generator axioms

```
fact {
  all t:Tree, f : Path | #t.content[f] <= 3
  one t : Tree | t.Inv and t.isEmpty
  all f : Path | some t : Tree |
      t.Inv and no t.Dirs and t.Files = f and no t.content[f]
  all f : Path, l : Line | some t : Tree |
      t.Inv and one t.Items and t.content[f] = (0 -> l)
  all f : Path, l1, l2 : Line | some t : Tree |
    t.Inv and one t.Items and t.content[f] = (0 -> l1) + (1 -> l2)
  all f : Path, l1, l2, l3: Line | some t : Tree |
    t.Inv and one t.Items and t.content[f] = (0 -> l1) + (1 -> l2) + (2 -> l3)
}
      -- 1 Path, 2 Line, max_file_size = 3
      -- 1 + (2^0 + 2^1 + 2^2 + 2^3) = 16
run {} for 16 but exactly 1 Path, exactly 2 Line, 0 Patch
```

- Mainly useful for verify operations on hunks.
- What happens if we change scope of Path from 1 to 2?
  - 31 trees instead of 16.
  - Number of variables/clauses explodes.

File size limit

```
assert Commute_EffectPreserving_Left {
  all p, q, q', p' : Hunk, t1, t2, t3 : Tree |
    (p.Inv and q.Inv and t1.Inv and
      Commute[p,q,q',p'] and Apply[t1,p,t2] and Apply[t2,q,t3])
        => some t2' : Tree | Apply[t1,q',t2'] and Apply[t2',p',t3]
}
```



Tree t1
File size limit = 3

Conclusions

- Alloy is not the right tool to verify Darcs.
  - Many limitations arised just trying to verify the "core of Darcs core".
  - Would be possible to do something useful when introducing sequences of patches?
- But anyway Alloy was useful to detect errors.
  - Errors writing the specification: too weak preconditions, stupid typos, ...
  - Darcs implementation errors: Int overflow, filesystem limits.

Understanding Darcs...

- Darcs User Manual.
- *Type-Correct Changes — A Safe Approach to Version Control Implementation.* Jason Dagit.
- *A Formalization of Darcs Patch Theory Using Inverse Semigroups.* Judah Jacobson.
- Several hours chatting with Ganesh Sittampalam (mainly), Ian Lynagh, Eric Kow, Petr Ročkai, Jason Dagit, ... (#darcs on FreeNode)
- 4 discussions on darcs-users@darcs.net.

Questions

Shoot!