Implementing the darcs patch formalism

...and verifying it

David Roundy

Cornell University

February 2006

The subject of this talk

Darcs a revision control system based on a formalism for manipulating changes, which allows for a system that is *change-based* rather than *version-based*. This talk will describe this formalism.

I will also describe a new trick using "Generalized Algebraic Data Types" (GADTs) to statically check the correctness of change-manipulation code.

Outline

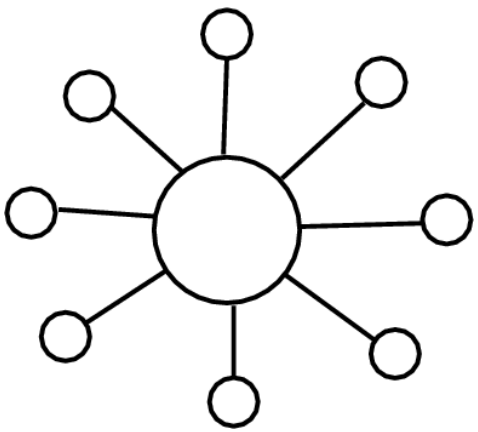
Darcs is a change-based revision control system, in contrast to the more common history-based revision control systems.

darcs

- has a friendly user interface
- uses an "egalitarian" distributed model
- allows "cherry picking" of changes
- avoids "merge points"—no history

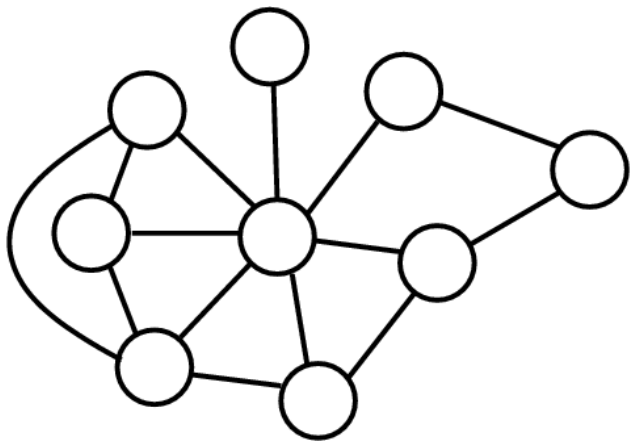
Distributed rather than centralized

Centralized



Examples: CVS, Subversion, Perforce

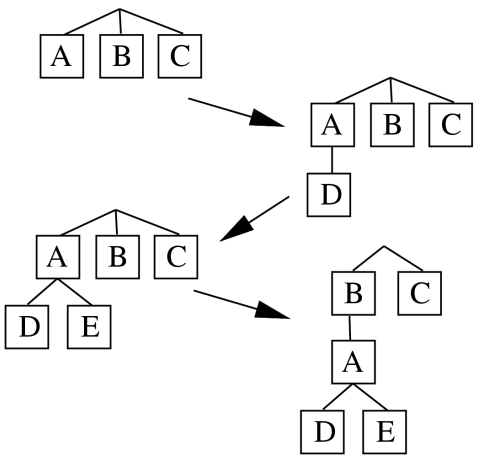
Distributed



Examples: darcs, Git, Bitkeeper, monotone, arch

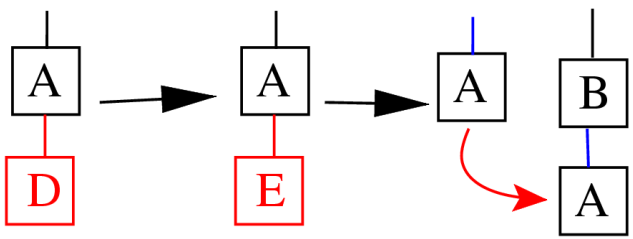
Change-based rather than version-based

Version-based



Examples: Git, Bitkeeper, Monotone, CVS, Subversion

Change-based



Examples: darcs

Darcs terminology

- A change is a logical entity.
- A patch is a description of a change.
- The state of a repository is defined by its set of changes.
- A set of changes is stored as a sequence of patches.

Notation

- A change is represented as a capital letter: A
- A patch is represented by a capital letter with possibly primes and/or a subscript: A, A', A_1
- Sometimes the state (or context) before and after a patch is represented by lowercase superscripts: ${}^{o}A^{a}$

The state of a repository is defined by a set of changes.

Patches are normally stored in sequence. For a sequential pair of patches, the final state of the first patch is identical to the initial state of the second patch.

Mathematical notation

$$ABC$$

or

$$^{o}A^{a}B^{b}C^{c}$$

.

Haskell notation

```
A :. B
```

or

```
A :- B :- C
```

.

Parallel and antiparallel

Parallel patches begin at the same state, and diverge to two different states, while antiparallel patches begin at different states and end at the same state. e.g. for the two patches:

$$^{o}A^{a} \quad \text{and} \quad ^{o}B^{b}$$

A is parallel to B and A^{-1} is antiparallel to B^{-1}.

Mathematical notation

$$A \vee B \quad \text{and} \quad A^{-1} \wedge B^{-1}$$

.

Haskell notation

```
A :\/: B   and   invert A :/\: invert B
```

.

Generalized Algebraic Data Types (GADTs)

"The solution to every problem is to create a new GADT."

"Generalized Algebraic Data Type"

- Also known as "guarded recursive data types" or "first-class phantom types"
- The common use example (which I won't give here) is to allow statically typesafe abstract syntax trees.
- Allow runtime behavior to statically restrict a subtype.

A *very* quick glance at Haskell syntax

- Types are capitalized, as in `Int`
- Functions are lowercase

```
foo :: Int -> Char -> Bool
```

`foo` is a function that accepts an `Int` and a `Char` as arguments, and returns a `Bool`

- Type *variables* are lowercase

```
bar :: a -> a -> a
```

`bar` is a function that accepts two arguments of any type, and returns a value of the same type.

Algebraic Data Types

```
data Bool where
  True :: Bool
  False :: Bool
```

```
data Complex where
  Cartesian :: Double -> Double -> Complex
  Polar :: Double -> Double -> Complex
  PureReal :: Double -> Complex
```

```
data Maybe a where
  Just :: a -> Maybe a
  Nothing :: Maybe a
```

Generalized Algebraic Data Types (GADTs)

```
data Pair a b where
  Pair :: a -> b -> Foo a b
  SymmetricPair :: a -> a -> Foo a a
```

The latter constructor restricts the type, which allows us to write typesafe code that wouldn't be possible with the more general type of a "`Pair a b`".

```
foo :: Pair a b -> b
foo (Pair x y) = y
foo (SymmetricPair x y) = x -- Note the oddness!
```

Phantom existential witness types

Phantom type

A type for which no data member is ever created. Most common example is the use in the ST monad to statically ensure that distinct states cannot be mixed.

Existential type

A type whose identity cannot be determined. The type is, however, known to exist, and may be known to have certain properties (e.g. be in a typeclass).

Witness type

A type whose existence is used to prove ("stand as witness") that something is true. Must be phantom.

Patch Properties

Inversion

Every darcs patch must be invertible.

Repercussions:

- A "remove file" patch must either contain the entire contents of the file, or one must only be able to remove a file after its contents have been removed. (darcs chooses the latter)
- A patch such as "copy file" is extra-complicated, since its inverse, a "merge two identical files" patch has confusing semantics (and thus the "copy file" patch would as well).
- We can apply patches either forwards or backwards to reach a particular version.
- Other benefits to be seen later when merging...

Inversion with phantom types as witnesses

The Haskell "Patch" type

```
data Patch a b where
    ...
```

This says that the `Patch` type is parametrized by two phantom types.

No GADTs yet, but we gain some expressiveness in function definitions:

Compare the Haskell code

```
invert :: Patch o a -> Patch a o
```

with the mathematical notation ${}^{o}A^{a}$ and ${}^{a}\left(A^{-1}\right)^{o}$.

Inverse of a sequence

The inverse of a sequence of patches is the sequence of their inverses, in reverse order.

$$(ABC)^{-1} = C^{-1}B^{-1}A^{-1}$$

```
x = invert (a :- b :- c)
y = invert c :- invert b :- invert a
-- x and y are the same...
```

Patch equality

- If two patches are equal, then both their representation, initial and final states are equal.
- Conversely, if two of these three are true, then the third must be also.

We need:

- A function that accepts two parallel patches and determines if they are equal by comparing their representation.
- A function that accepts two anti-parallel patches and determines if they are equal by comparing their representation.

Note: Checking the representation *alone* is not enough to guarantee equality, since non-equal patches may have the same representation when expressed in different contexts (e.g. "remove the first line of a file").

GADT witnesses and patch equality

GADT as witness of type equality

```
data EqCheck a b where
    NotEq :: EqCheck a b
    IsEq :: EqCheck a a
```

Two equality check operators

```
(=\/=) :: Patch a b -> Patch a c -> EqCheck b c
(=/\=) :: Patch a c -> Patch b c -> EqCheck a b
    -- Implemented using unsafeCoerce#
```

```
example ((a :- b) :\/: (a' :- c)) =
    case a =\/= a' of
    IsEq -> example2 (b :\/: c)
    ...
```

$$(AB) \vee (A'C)$$

$$A = A'$$

$$B \vee C$$

Commutation

Commutation is both a relationship and a function, which reorders a pair of sequential patches. Commutation may fail.

Mathematical notation

$$AB \leftrightarrow B'A' \qquad \text{or} \qquad {}^{o}A^{a}B^{b} \leftrightarrow^{o} B'^{x}A'^{b}$$

Haskell notation

```
commute :: Sequential o b -> Maybe (Sequential o b)

... do (a :- b :- c) <- abc
       b' :. a' <- commute (a :. b)
       c' :. a'' <- commute (a' :. c)
```

Commutation is self-inverting

Commutation—when successful—is self-inverting.

```
verify_commute (a :. b) | isJust (commute (a :. b)) =
  isJust $ do b1 :. a1 <- commute (a :. b)
              a' :. b' <- commute (b1 :. a1)
              IsEq <- a' =\/= a
              IsEq <- b' =/\= b -- could use =\/=
```

Commutation of an inverse sequential pair

Commutation with the inverse of a sequential pair gives the same result as the inverse of the commutation of the pair.

$$AB \leftrightarrow B_1 A_1$$
$$B^{-1}A^{-1} \leftrightarrow A_1^{-1}B_1^{-1}$$

```
verify_commute (a :. b) | isJust (commute (a :. b)) =
    isJust $
    do b1 :. a1 <- commute (a :. b)
       ia1 :. ib1 <- commute (invert b :. invert a)
       IsEq <- b1 =\/= invert ib1
       IsEq <- a1 =\/= invert ia1
```

Commutation with patch and its inverse

Commutation with a patch and its inverse, if successful, does not alter a patch. If the first commute is successful, then the other must be also.

$$AB \leftrightarrow B_1 A_1$$
$$A^{-1} B_1 \leftrightarrow B A_1^{-1}$$

```
verify_commute (a :. b) | isJust (commute (a :. b)) =
    isJust $ do b1 :. a1 <- commute (a :. b)
                b' :. ia1 <- commute (invert a :. b1)
                IsEq <- b' =\/= b
                IsEq <- invert ia1 =\/= a1
```

Permutivity

Permutivity is the property of the commute that means that any commuted permutation is uniquely defined, regardless of the order of commutation.

$$ABC \leftrightarrow AC_1B_1 \leftrightarrow C_2A_1B_1 \leftrightarrow C_2B_2A_2 \leftrightarrow B_3C_3A_2 \leftrightarrow B_3A_3C$$

- Ganesh has proven that if permutivity holds for *any* set of sequence of three patches, then it holds for any sequence of N patches.
- Only applies to permutations that exist

Application: a merge

A merge is an operation that takes two parallel patches, and converts them into a pair of sequential patches.
Commutation of the sequential pair must allow recovery of both original patches.

The merge of $A \vee B$ is

$$AB_1 \leftrightarrow BA_1$$

Using the property of commutation with a patch and its inverse:

$$B^{-1}A \leftrightarrow A_1 B_1^{-1}$$

which allows us to compute the merged result using only the merge and invert functions.

Application: a merge

The merge of $A \vee B$ is

$$AB_1 \leftrightarrow BA_1$$

Using the property of commutation with a patch and its inverse:

$$B^{-1}A \leftrightarrow A_1 B_1^{-1}$$

which allows us to compute the merged result using only the merge and invert functions.

```
merge :: Parallel a b -> Maybe (AntiParallel b a)
-- Input is A and B, output is A_1 and B_1
merge (a :\/: b) =
    do a1 :. ib1 <- commute (invert b :. a)
       return (a1 :/\: invert ib1)
```

Conclusions

"The solution to every problem is to create a new GADT."

- Patch manipulation is lots of fun.
- GADTs are also lots of fun.
- Witness types allow us to prevent large classes of bugs.

Interested darcs users:

Meet at 1:00 outside this building for a brainstorming discussion of what features really ought to be added to darcs.