Structured diffs: theory and practice

ICFP PC @ SLC

Victor Cacciari Miraldo, Pierre-Evariste Dagand,
Giovanni Garufi, Marco Vassena and Wouter Swierstra

The `diff` utility

The Unix `diff` utility compares two files line-by-line, computing the smallest number of insertions and deletions to transform one into the other.

It was developed as far back as 1976 – but still forms the heart of many modern version control systems such as git, mercurial, svn, and many others.

Example: comparing two files

`slc-teams.csv`

Real Salt Lake, Soccer

Utah Jazz, Basketball

Salt Lake Bees, Baseball

Example: comparing two files

`slc-teams.csv`

Real Salt Lake, Soccer

Utah Jazz, Basketball

Salt Lake Bees, Baseball

`slc-teams-fixed.csv`

Real Salt Lake, **Football**

Utah Jazz, Basketball

Salt Lake Bees, Baseball

Example: comparing two files

```
-Real Salt Lake, Soccer
+Real Salt Lake, Football
Utah Jazz,  Basketball
Salt Lake Bees,  Baseball
```

The `diff` utility computes a *patch*, that can be used to transform the one file into the other.

Smallest edit script

Crucially, `diff` always computes the **smallest** patch – minimizing the number of insertions and deletions.

Smallest edit script

Crucially, `diff` always computes the **smallest** patch – minimizing the number of insertions and deletions.

In other words, it tries to preserve as much information as possible.

Smallest edit script

Crucially, `diff` always computes the **smallest** patch – minimizing the number of insertions and deletions.

In other words, it tries to preserve as much information as possible.

But sometimes it still doesn't do a very good job.

Example: comma separated values

slc-teams-fixed.csv

```
Real Salt Lake, Football
Utah Jazz,  Basketball
Salt Lake Bees,  Baseball
```

How would this file change if I add a new column?

Example: comma separated values

```
-Real Salt Lake, Football
+Real Salt Lake, Football, 2004
-Utah Jazz,     Basketball
+Utah Jazz,     Basketball, 1979
-Salt Lake Bees,  Baseball
+Salt Lake Bees,  Baseball, 1994
```

Example: comma separated values

```
-Real Salt Lake, Football
+Real Salt Lake, Football, 2004
-Utah Jazz,     Basketball
+Utah Jazz,     Basketball, 1979
-Salt Lake Bees,  Baseball
+Salt Lake Bees,  Baseball, 1994
```

Adding a new column changes **every line** in our original file.

Where conceptually, we are not modifying any existing **data**.

Example: comma separated values

```
-Real Salt Lake, Football
+Real Salt Lake, Football, 2004
-Utah Jazz,     Basketball
+Utah Jazz,     Basketball, 1979
-Salt Lake Bees,  Baseball
+Salt Lake Bees,  Baseball, 1994
```

Adding a new column changes **every line** in our original file.

Where conceptually, we are not modifying any existing **data**.

Not all data is best represented by a list of lines!

This is particularly important when using `diff` to compare *source code*.

What is the diff over structured data?

Questions

- How can we represent a family of data types?
- How can we represent patches on these data types?
- Does this give a better account of software evolution?

Questions

- **How can we represent a family of data types?**
- How can we represent patches on these data types?
- Does this give a better account of software evolution?

Universe of discourse

We will use Agda as our metalanguage to answer these questions and start by fixing a ‘sums of products’ universe:

```
data Atom : Set where
  K : U -> Atom
  I : Atom

Prod : Set
Prod = List Atom

Sum : Set
Sum = List Prod
```

Here we assume some ‘base universe’ `U`, storing the atomic types such as integers, characters, etc.

Semantics

We can interpret these types as *pattern functors*:

```
elA : Atom -> (Set -> Set)
elA I       X   = X
elA (K u)   X   = elU u

elP : Prod -> (Set -> Set)
elP []            X   = Unit
elP (a :: as)  X   = Pair (elA alpha X) (elP pi X)

elS : Sum -> (Set -> Set)
elS []          X   = Empty
elS (p :: ps) X   = Either (elP p X) (elS ps X)
```

Fixpoints

Given any element of our ‘sums of products’ universe, we can compute the corresponding pattern functor.

Taking the least fixpoint of this functor allows us to tie the recursive knot:

```
data Fix (s : Sum) : Set where
  <_> : elS s (Fix s) -> Fix s
```

Example: 2-3 trees

We can represent 2-3-trees defined as follows:

```
data Tree : Set where
  leaf    : Tree
  2-node  : Nat -> Tree -> Tree -> Tree
  3-node  : Nat -> Tree -> Tree -> Tree -> Tree
```

by the following sum-of-products:

```
tree23F  : Sum
tree23F =  let leafT  = []
               node2T = [ K NAT , I , I ]
               node3T = [ K NAT , I , I , I ]
           in [leafT , node2T , node3T ]
```

Questions

- How can we represent a family of data types?
- **How can we represent patches on these data types?**
- Does this give a better account of software evolution?

2-3-trees

```
treeA = 2-node 7 t1 t2

treeB = 3-node 12 (2-node 7 t1 leaf) leaf leaf
```

What edit script should transform `treeA` to `treeB`?

2-3-trees

```
treeA = 2-node 7 t1 t2

treeB = 3-node 12 (2-node 7 t1 leaf) leaf leaf
```

What edit script should transform `treeA` to `treeB`?

It is not just a list of insertions and deletions!

We can insert new constructors, modify values stored in the tree, delete subtrees, or copy over existing data.

We will use a *type indexed data type* to account for changes.

Representing diffs

Our universe consists of three separate layers:

- sums
- products
- atomic values

We'll define what it means to modify each of these layers – from these pieces we can define our overall type for diffs.

Spines: changes to sums

Given two arbitrary tree structures, `x` and `y`, we can identify the following three cases:

1. `x` and `y` are equal;
2. `x` and `y` the same outermost constructor, but are not equal trees;
3. `x` and `y` have a different outermost constructor.

Spines: changes to sums

Given two arbitrary tree structures, `x` and `y`, we can identify the following three cases:

1. `x` and `y` are equal;
2. `x` and `y` the same outermost constructor, but are not equal trees;
3. `x` and `y` have a different outermost constructor.

To represent patches, we need a data type that describes these three cases.

But what information should each constructor record?

Spines

Assuming that we know what patches on atoms (`pAt`) and products (`pAl`) are we can define:

```
data S (σ : Sum) : Set where
  Scp   : S σ
  Scns  : (C : Constr σ)
        -> All pAt (fields C)
        -> S σ
  Schg  : (C1 C2 : Constr σ)
        -> pAl (fields C1) (fields C2)
        -> S σ
```

We still need to define how to diff **products** and **atoms**.

Alignments: changes to products

If we have reconciled the choice of constructor, how to we compare the constructor fields?

Alignments: changes to products

If we have reconciled the choice of constructor, how to we compare the constructor fields?

Each value constructed in our universe has a *list of fields* – the product structure.

Given two such lists, we need to compare them somehow.

Yet these fields may store values of very different types!

Alignments: changes to products

If we have reconciled the choice of constructor, how to we compare the constructor fields?

Each value constructed in our universe has a *list of fields* – the product structure.

Given two such lists, we need to compare them somehow.

Yet these fields may store values of very different types!

The good news, however, is that we can reuse ideas from the classic `diff` algorithm at this point.

Alignments: changes to products

To describe a change from one list of constructor fields to another, we require an *edit script* that:

- copies over fields;
- deletes fields;
- inserts new fields.

Alignments

```
data Al : Prod → Prod → Set where
  A0 : Al At [] []
  AX : At α → Al π2 π1 → Al At (α :: π2) (α :: π1)
  Adel : elA a → Al π2 π1 → Al (α :: π2) π1
  Ains : elA a → Al π2 π1 → Al π2 (α :: π1)
```

A value of type `Al π2 π1` prescribes which fields of one constructor are matched with which fields of another.

Atoms

Finally, we still need to handle our atomic values.

For constant types, we can check if they are equal or not.

Atoms

Finally, we still need to handle our atomic values.

For constant types, we can check if they are equal or not.

But what about recursive subtrees?

Handling recursive data types

So far our *spines* compare the outermost constructors.

Oftentimes, you may want to delete certain constructors (exposing its subtrees) or insert new constructors.

We cannot handle such changes with the data types we have seen so far…

Accounting for recursion

Our final patch type identifies three cases:

1. The insertion of a new constructor, together with all-but-one of its fields;
2. The deletion of the outermost constructor, together with all-but-one of its fields;
3. A choice of spine, alignment, and a patch on atomic values;

The first two require additional information – a context – to point out *where* to insert/delete a subtree.

Applying patches

We can define generic operations – such as patch application – that applies a patch to a given tree:

```
apply : Patch → Fix σ → Maybe (Fix σ)
```

This patch is guaranteed to **preserve types**.

It may still fail – when encountering an unexpected constructor or atomic value – but it will never produce ill-formed data.

Questions

- How can we represent a family of data types?
- How can we represent patches on these data types?
- **Does this give a better account of software evolution?**

Case study: Clojure

- We've instantiated this algorithm to a simplified Clojure AST in Haskell;
- By implementing a simple Clojure parser, we can now compare Clojure programs.

Case study: Clojure

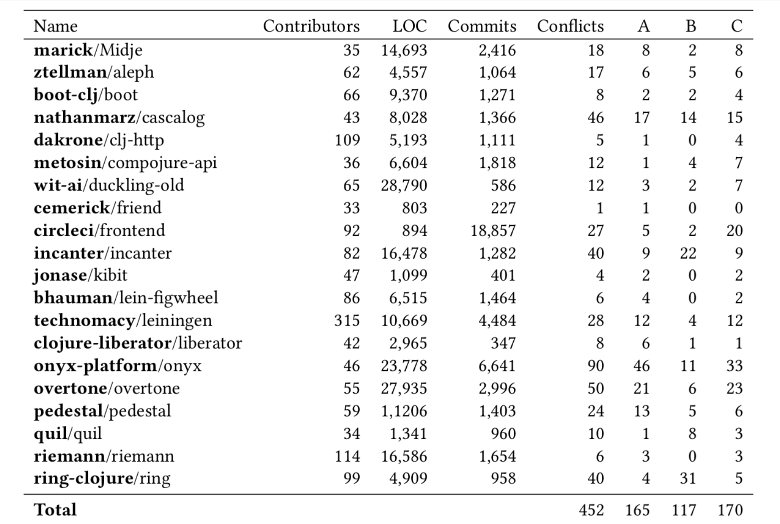
- We've instantiated this algorithm to a simplified Clojure AST in Haskell;
- By implementing a simple Clojure parser, we can now compare Clojure programs.
- And by mining the commit history of the top Clojure repositories on GitHub, we can try to quantify the performance our algorithm.

Collect data

A) *False conflicts* – two changes to the same line that do not overlap in the AST

B) *Fixable conflicts* – two changes to the same atomic value, where knowing the abstract syntax tree allows us to resolve them automatically/interactively.

C) *True conflicts* – two atomic values (integers, variables, etc.) changed in different ways



Name	Contributors	LOC	Commits	Conflicts	A	B	C
marick/Midje	35	14,693	2,416	18	8	2	8
ztellman/aleph	62	4,557	1,064	17	6	5	6
boot-clj/boot	66	9,370	1,271	8	2	2	4
nathanmarz/cascalog	43	8,028	1,366	46	17	14	15
dakrone/clj-http	109	5,193	1,111	5	1	0	4
metosin/compojure-api	36	6,604	1,818	12	1	4	7
wit-ai/duckling-old	65	28,790	586	12	3	2	7
cemerick/friend	33	803	227	1	1	0	0
circleci/frontend	92	894	18,857	27	5	2	20
incanter/incanter	82	16,478	1,282	40	9	22	9
jonase/kibit	47	1,099	401	4	2	0	2
bhauman/lein-figwheel	86	6,515	1,464	6	4	0	2
technomacy/leiningen	315	10,669	4,484	28	12	4	12
clojure-liberator/liberator	42	2,965	347	8	6	1	1
onyx-platform/onyx	46	23,778	6,641	90	46	11	33
overtone/overtone	55	27,935	2,996	50	21	6	23
pedestal/pedestal	59	1,1206	1,403	24	13	5	6
quil/quil	34	1,341	960	10	1	8	3
riemann/riemann	114	16,586	1,654	6	3	0	3
ring-clojure/ring	99	4,909	958	40	4	31	5
Total				452	165	117	170

Results

Interpreting these results

- Conflicts are rare! 452 conflicts found in tens of thousands of commits.

Interpreting these results

- Conflicts are rare! 452 conflicts found in tens of thousands of commits.

Or perhaps hard to observe as rebasing can rewrite history, complicated pull requests abandoned, etc.

- Structure aware algorithms can beat line-based diff

Interpreting these results

- Conflicts are rare! 452 conflicts found in tens of thousands of commits.

Or perhaps hard to observe as rebasing can rewrite history, complicated pull requests abandoned, etc.

- Structure aware algorithms can beat line-based diff

But performance of our algorithm is still lagging behind.

Questions?