Structured diffs: theory and practice
ICFP PC @ SLC
Victor Cacciari Miraldo, Pierre-Evariste Dagand, Giovanni Garufi, Marco Vassena and Wouter Swierstra
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.
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:

```agda
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:

\[
\begin{align*}
elA : \text{Atom} & \rightarrow (\text{Set} \rightarrow \text{Set}) \\
elA \ I \ X & = X \\
elA (K \ u) \ X & = elU \ u \\
elP : \text{Prod} & \rightarrow (\text{Set} \rightarrow \text{Set}) \\
elP [] \ X & = \text{Unit} \\
elP (a :: as) \ X & = \text{Pair} (elA \ alpha \ X) (elP \ pi \ X) \\
elS : \text{Sum} & \rightarrow (\text{Set} \rightarrow \text{Set}) \\
elS [] \ X & = \text{Empty} \\
elS (p :: ps) \ X & = \text{Either} (elP \ p \ X) (elS \ ps \ X)
\end{align*}
\]
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:

```haskell
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

\[
\text{treeA} = \text{2-node 7 t1 t2}
\]

\[
\text{treeB} = \text{3-node 12 (2-node 7 t1 leaf) leaf leaf}
\]

What edit script should transform \text{treeA} to \text{treeB}?
2-3-trees

\[
\text{treeA} = \text{2-node } 7 \ t1 \ t2
\]

\[
\text{treeB} = \text{3-node } 12 \ (\text{2-node } 7 \ t1 \ \text{leaf}) \ \text{leaf} \ \text{leaf}
\]

What edit script should transform \text{treeA} to \text{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.
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?
Assuming that we know what patches on atoms (pAt) and products (pAl) are we can define:

```haskell
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.
If we have reconciled the choice of constructor, how do 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
  Aθ : 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 $\text{Al} \; \pi_2 \; \pi_1$ 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.
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...
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:

\[
\text{apply} : \text{Patch} \rightarrow \text{Fix } \sigma \rightarrow \text{Maybe } (\text{Fix } \sigma)
\]

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

| Total                 |              | 452   | 165     | 117       | 170|

Results

Universiteit Utrecht
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?