Type-directed Diffing of Structured Data

Victor Cacciari Miraldo, Pierre-Évariste Dagand
and Wouter Swierstra

January 5, 2018

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

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

jabber.txt

```
Twas brillig, and the slithy toves
Waved to Mars, where a robot roves;
Did gyre and gimble in the wabe;
And the mome raths outgrabe.
```

wocky.txt

```
Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.
```

Example: comparing two files

```
  Twas brillig, and the slithy toves
- Waved to Mars, where a robot roves;
  Did gyre and gimble in the wabe;
+ All mimsy were the borogoves,
  And the mome raths outgrabe.
```

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.

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.

Tries to preserve as much information as possible.

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

Example: comma separated values

bibliography.csv

```
Lewis Carroll, The alphabet cipher
Lewis Carroll, The game of logic
Lewis Carroll, The hunting of the snark
```

How would this file change if I add publication dates?

Example: comma separated values

```
- Lewis Carroll, The alphabet cipher
+ Lewis Carroll, The alphabet cipher, 1868
- Lewis Carroll, The game of logic
+ Lewis Carroll, The game of logic, 1887
- Lewis Carroll, The hunting of the snark
+ Lewis Carroll, The hunting of the snark, 1876
```

Example: comma separated values

```
- Lewis Carroll, The alphabet cipher
+ Lewis Carroll, The alphabet cipher, 1868
- Lewis Carroll, The game of logic
+ Lewis Carroll, The game of logic, 1887
- Lewis Carroll, The hunting of the snark
+ Lewis Carroll, The hunting of the snark, 1876
```

Syntatically changes **every line**.

Semantically, **data** was not modified.

Example: comma separated values

```
- Lewis Carroll, The alphabet cipher
+ Lewis Carroll, The alphabet cipher, 1868
- Lewis Carroll, The game of logic
+ Lewis Carroll, The game of logic, 1887
- Lewis Carroll, The hunting of the snark
+ Lewis Carroll, The hunting of the snark, 1876
```

Syntatically changes **every line**.

Semantically, **data** was not modified.

Particularly important when `diff`'ing source code.

What is the diff over structured data?

Questions

- How can we represent data types?
- How can we represent patches on these data types?
- How can we compute such patches?
- How can we merge such patches?

Questions

- **How can we represent data types?**
- How can we represent patches on these data types?
- How can we compute such patches?
- How can we merge such patches?

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:

$$
\begin{aligned}
&[\![\cdot]\!]_{\mathsf{a}} : \mathsf{Atom} \to (\mathsf{Set} \to \mathsf{Set})\\
&[\![\mathsf{I}]\!]_{\mathsf{a}}\ X = X\\
&[\![\mathsf{K}\ \kappa]\!]_{\mathsf{a}}\ X = [\![\kappa]\!]_{\mathsf{k}}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\cdot]\!]_{\mathsf{p}} : \mathsf{Prod} \to (\mathsf{Set} \to \mathsf{Set})\\
&[\![[]]\!]_{\mathsf{p}}\ X = \mathsf{Unit}\\
&[\![a :: as]\!]_{\mathsf{p}}\ X = [\![\alpha]\!]_{\mathsf{a}}\ X \times [\![\pi]\!]_{\mathsf{p}}\ X
\end{aligned}
$$

$$
\begin{aligned}
&[\![\cdot]\!]_{\mathsf{s}} : \mathsf{Sum} \to (\mathsf{Set} \to \mathsf{Set})\\
&[\![[]]\!]_{\mathsf{s}}\ X = \bot\\
&[\![p :: ps]\!]_{\mathsf{s}}\ X = [\![p]\!]_{\mathsf{p}}\ X \uplus [\![ps]\!]_{\mathsf{s}}\ X
\end{aligned}
$$

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

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:

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Fix}\ (s\ :\ \mathsf{Sum})\ :\ \mathsf{Set}\ \mathsf{where} \\
\quad \langle\cdot\rangle\ :\ [\![s]\!]_{\mathsf{s}}\ (\mathsf{Fix}\ s)\ \to\ \mathsf{Fix}\ s
\end{array}
$$

Example: 2-3 trees

We can represent a 2-3-tree, usually defined as follows:

$$
\begin{array}{lll}
\textsf{data Tree} & : & \textsf{Set where} \\
\quad \textsf{leaf} & : & \textsf{Tree} \\
\quad \textsf{2-node} & : & \mathbb{N} \to \textsf{Tree} \to \textsf{Tree} \to \textsf{Tree} \\
\quad \textsf{3-node} & : & \mathbb{N} \to \textsf{Tree} \to \textsf{Tree} \to \textsf{Tree} \to \textsf{Tree}
\end{array}
$$

by the following sum-of-products:

$$
\begin{array}{lll}
\textsf{Tree}_F & : & \textsf{Sum} \\
\textsf{Tree}_F & = & \textsf{let}\ leafT \quad = [] \\
 & & \qquad node2T = [\ \textsf{K}\ \mathbb{N}\ ,\ \textsf{I}\ ,\ \textsf{I}\ ] \\
 & & \qquad node3T = [\ \textsf{K}\ \mathbb{N}\ ,\ \textsf{I}\ ,\ \textsf{I}\ ,\ \textsf{I}\ ] \\
 & & \textsf{in}\ [leafT\ ,\ node2T\ ,\ node3T\ ]
\end{array}
$$

Questions

- How can we represent data types?
- **How can we represent patches on these data types?**
- How can we compute such patches?
- How can we merge such patches?

2-3-trees

$$\text{treeA} \; = \; \text{2-node } 7 \; t_1 \; t_2$$

$$\text{treeB} \; = \; \text{3-node } 12 \; (\text{2-node } 7 \; t_1 \text{ leaf}) \text{ leaf leaf}$$

What edit script transforms treeA into treeB?

2-3-trees

$$\text{treeA} = \text{2-node } 7 \; t_1 \; t_2$$
$$\text{treeB} = \text{3-node } 12 \; (\text{2-node } 7 \; t_1 \; \text{leaf}) \; \text{leaf leaf}$$

What edit script transforms treeA into 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.

Representing diffs

We define a type indexed data type, to account for changes, defining what it means to modify each layer of our universe.

- sums
- products
- atomic values

From these pieces we define our overall type for diffs.

Spines: changes to sums

Given two arbitrary tree structures, `x` and `y`, either:

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`, either:

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, S, capture these three cases.

Spines

Assuming we know what patches on atoms (At) and products (Al) are, we define:

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{S}\ (\sigma\ :\ \mathsf{Sum})\ :\ \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{Scp}\ \ :\ \mathsf{S}\ \sigma \\
\quad \mathsf{Scns}\ :\ (C\ :\ \mathsf{Constr}\ \sigma) \\
\qquad\qquad \to\ \mathit{All}\ \mathsf{At}\ (\mathsf{fields}\ C) \\
\qquad\qquad \to\ \mathsf{S}\ \sigma \\
\quad \mathsf{Schg}\ :\ (C_1\ C_2\ :\ \mathsf{Constr}\ \sigma) \\
\qquad\qquad \to\ \mathsf{Al}\ (\mathsf{fields}\ C_1)\ (\mathsf{fields}\ C_2) \\
\qquad\qquad \to\ \mathsf{S}\ \sigma
\end{array}
$$

Next we define the diff for **products** and **atoms**.

Alignments: changes to products

How to compare fields of reconciled constructors? (Schg case)

Alignments: changes to products

How to compare fields of reconciled constructors? (Schg case)

Each value has a list of fields – the product structure.

These fields can have different types!

Alignments: changes to products

How to compare fields of reconciled constructors? (Schg case)

Each value has a list of fields – the product structure.

These fields can have different types!

Good news: Unix `diff` algorithm computes a diff for lists of lines.

Alignments: changes to products

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

- changes one field into another;
- deletes fields;
- inserts new fields.

Alignments

$$
\begin{array}{lll}
\mathsf{data}\ \mathsf{Al} & : & \mathsf{Prod} \rightarrow \mathsf{Prod} \rightarrow \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{A0} & : & \mathsf{Al}\ []\ [] \\
\quad \mathsf{AX} & : & \mathsf{At}\ \alpha \rightarrow \mathsf{Al}\ \pi_1\ \pi_2 \rightarrow \mathsf{Al}\ (\alpha :: \pi_1)\ (\alpha :: \pi_2) \\
\quad \mathsf{Adel} & : & [\![\alpha]\!]_{\mathsf{a}} \rightarrow \mathsf{Al}\ \pi_1\ \pi_2 \rightarrow \mathsf{Al}\ (\alpha :: \pi_1)\ \pi_2 \\
\quad \mathsf{Ains} & : & [\![\alpha]\!]_{\mathsf{a}} \rightarrow \mathsf{Al}\ \pi_1\ \pi_2 \rightarrow \mathsf{Al}\ \pi_1\ (\alpha :: \pi_2)
\end{array}
$$

A value of type $\mathsf{Al}\ \pi_1\ \pi_2$ indicates which fields of one constructor are matched with which fields of another.

Analogous to UNIX `diff` and lines.

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, one wants 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. Insertion of a constructor, with a zipper over its fields;
2. Deletion the outermost constructor, with a zipper over its fields;
3. A choice of spine, alignment, and a patch on atomic values;

The first two carry that zipper to point out where to insert/delete a subtree. We call this the context.

Applying patches

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

$$\mathsf{apply} : \mathsf{Patch} \rightarrow \mathsf{Fix}\,\sigma \rightarrow \mathsf{Maybe}\,(\mathsf{Fix}\,\sigma)$$

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 data types?
- How can we represent patches on these data types?
- **How can we compute such patches?**
- How can we merge such patches?

Computing Patches

Many patches transform one x into a y.

Computing Patches

Many patches transform one x into a y.

Too expensive to enumerate. Counting copies not enough.

Computing Patches

Many patches transform one x into a y.

Too expensive to enumerate. Counting copies not enough.

Heuristics to prune the search space.

- UNIX `diff3`.
- Edit scripts preorder traversal.
- ...

Computing Patches

Many patches transform one x into a y.

Too expensive to enumerate. Counting copies not enough.

Heuristics to prune the search space.

- UNIX `diff3`.
- Edit scripts preorder traversal.
- ...

Implemented by the means of Oracles.

Computing Patches: Oracles

$$
\begin{tikzcd}
\mathsf{Fix}\ \sigma \times \mathsf{Fix}\ \sigma \arrow[r, "\mathsf{enum}"] & \mathsf{List\ Patch} \arrow[d, "\mathsf{best}"] \\
 & \mathsf{Patch}
\end{tikzcd}
$$

Computing Patches: Oracles

$$
\begin{tikzcd}
\mathsf{Fix}\ \sigma \times \mathsf{Fix}\ \sigma \arrow[r, "\mathsf{enum}"] \arrow[d, "\mathcal{O}"'] & \mathsf{List\ Patch} \arrow[d, "\mathsf{best}"] \\
\mathsf{Fix}_a\ \sigma \times \mathsf{Fix}_a\ \sigma \arrow[r, "\mathsf{tr}"'] & \mathsf{Patch}
\end{tikzcd}
$$

Flag indicating copy:

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Fix}_a\ (s\ :\ \mathsf{Sum})\ :\ \mathsf{Set}\ \mathsf{where} \\
\quad \langle \cdot, \cdot \rangle\ :\ \mathsf{Bool} \rightarrow [\![s]\!]_{\mathsf{s}}\ (\mathsf{Fix}_a\ s) \rightarrow \mathsf{Fix}_a\ s
\end{array}
$$

Computing Patches: Oracles

$$
\begin{tikzcd}
\mathsf{Fix}\ \sigma \times \mathsf{Fix}\ \sigma \arrow[r, "\mathsf{enum}"] \arrow[d, "\mathcal{O}"'] & \mathsf{List\ Patch} \arrow[d, "\mathsf{best}"] \\
\mathsf{Fix}_a\ \sigma \times \mathsf{Fix}_a\ \sigma \arrow[r, "\mathsf{tr}"'] & \mathsf{Patch}
\end{tikzcd}
$$

Flag indicating copy:

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Fix}_a\ (s : \mathsf{Sum}) : \mathsf{Set}\ \mathsf{where} \\
\quad \langle \cdot , \cdot \rangle : \mathsf{Bool} \to [\![s]\!]_{\mathsf{s}}\ (\mathsf{Fix}_a\ s) \to \mathsf{Fix}_a\ s
\end{array}
$$

Domain specific. The better the Oracle, the better the resulting patch.

Questions

- How can we represent data types?
- How can we represent patches on these data types?
- How can we compute such patches?
- **How can we merge such patches?**

Merging Square

Merging disjoint patches trivially commutes:

$$\begin{tikzcd} & x \arrow[dl, "p"'] \arrow[dr, "q"] & \\ y \arrow[dr, "\textsf{merge}\ p\ q"'] & & y' \arrow[dl, "\textsf{merge}\ q\ p"] \\ & z & \end{tikzcd}$$

Merging Square

Merging disjoint patches trivially commutes:

$$
\begin{tikzcd}
 & x \arrow[dl, "p"'] \arrow[dr, "q"] & \\
y \arrow[dr, "\mathsf{merge}\ p\ q"'] & & y' \arrow[dl, "\mathsf{merge}\ q\ p"] \\
 & z &
\end{tikzcd}
$$

That is,

$$\mathsf{apply}\ (\mathsf{merge}\ p\ q) \circ \mathsf{apply}\ p \equiv \mathsf{apply}\ (\mathsf{merge}\ q\ p) \circ \mathsf{apply}\ q$$

Related work

- There is a great deal of work on comparing (untyped) tree comparisons – but much less work that attempts to exploit the type structure that we have available.
- Lempsink et al. & Vassena are a notable exception – but run a linear diff on the traversal of the tree. This it hard to guarantee that later operations – such as merging patches – produce well-formed trees.

Looking ahead

Work in Progress

- Prove properties about the efficient route of the "computing patches" square.
- Empirical Validation: Analisys of Clojure data from GitHub repos. Conducted by our MSc Giovanni Garufi.

Future Work

- Implement a *proof-of-concept* in Haskell.
- Incorporate conflicts to our model.

Type-directed Diffing of Structured Data

Victor Cacciari Miraldo, Pierre-Évariste Dagand
and Wouter Swierstra

January 5, 2018