Type directed diffing of structured data

Victor Cacciari Miraldo, Pierre-Evariste Dagand and Wouter Swierstra

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.

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

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
```

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

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

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
```

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?
- How can we compute a patch between two values?

Questions

- **How can we represent a family of data types?**
- How can we represent patches on these data types?
- How can we compute a patch between two values?

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?**
- How can we compute a patch between two values?

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 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.

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.

The last point is quite subtle: all our definitions of spines, alignments and atomic patches were parametrized by how to handle recursive occurrences – here we tie the recursive knot!

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?
- **How can we compute a patch between two values?**

Computing patches

There may be many different patches, transforming one value into another.

The `diff` utility has a clear definition of 'best' patch: always choose the patch with the least number of deletions/insertions.

This works because every line is assumed to have comparable length; lines are never nested.

Computing patches

There may be many different patches, transforming one value into another.

The `diff` utility has a clear definition of ‘best’ patch: always choose the patch with the least number of deletions/insertions.

This works because every line is assumed to have comparable length; lines are never nested.

It is not so clear how to generalize this:

- deleting/inserting large subtrees should be expensive;
- but many small modifications may sometimes be worse than deleting/inserting a larger subtree.

Enumerating patches

Rather than fix one particular choice up front, we chose to **enumerate** all possible patches between two trees.

You may want to think of this as a non-deterministic program – later heuristics or user-interaction might help to select the 'best' patch.

Computing spines & alignments (sketch)

- Given two trees, computing the corresponding *spine* is deterministic. We only need to compare the trees and their outermost constructor.
- Given two lists of constructor fields, there are many different *alignments*:
- if the source list is empty, insert all remaining values;
- if the target list is empty, delete all remaining values;
- otherwise, consider performing a single insertion/deletion/modification to the current fields.

We can extend this to handle the addition or removal of constructors.

Complexity

This algorithm is inpractical for any non-trivial trees – but it defining it enables further exploration!

For example, we may want to define *domain specific heuristics* to prune the search space:

- try to line up top-level functions with the same in a file;
- avoid deleting certain constructors;
- insert other constructors whenever possible;
- consult an external oracle (such as a user or the output of `diff`) to guide the process.

We illustrate our ideas with a simple example in the paper.

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

- We've started instantiating our ideas to specific data types – such as a (simplified) AST for Clojure. Will our diff be more accurate on existing code?
- We'd like to enrich our universe further to cover richer (dependent) types and account for variable binding.
- We would like to describe how to *merge* two independent patches, incorporating changes from both.

Discussant

Discussant

Questions?