Efficient Structural Differencing

Victor Cacciari Miraldo   Wouter Swierstra

Utrecht University

Why Structural Differencing?

Motivation

```
Flour , B5, 5

Sugar , B7, 12

...
```

Motivation

```
Flour , B5, 5

Sugar , B7, 12

...
```

```
Flour , B5, 5

Sugar , F0, 12

...
```

Motivation

```
Flour , B5, 5
Sugar , B7, 12
...
```

```
Flour , B5, 5
Sugar , F0, 12
...
```

```
Flour , B5, 5
Sugar , B7, 42
...
```

Motivation

```
Flour , B5, 5
Sugar , B7, 12
...
```

```
Flour , B5, 5
Sugar , F0, 12
...
```

```
Flour , B5, 5
Sugar , B7, 42
...
```

Same line changes in two different ways

Motivation

```
Flour , B5, 5          Flour , B5, 5          Flour , B5, 5
Sugar , B7, 12         Sugar , F0, 12         Sugar , B7, 42
...                    ...                    ...
```

Same line changes in two different ways

Not same *column*

Motivation

```
Flour , B5, 5          Flour , B5, 5          Flour , B5, 5
Sugar , B7, 12         Sugar , F0, 12         Sugar , B7, 42
...                    ...                    ...
```

Same line changes in two different ways

Not same *column*

Here, merging requires knowledge about structure

Contributions

Contributions

- Representation for changes

Contributions

- Representation for changes
- Efficient Algorithm for structured diffing (and merging)
    - Think of `UNIX` diff, over algebraic datatypes.

Contributions

- Representation for changes

- Efficient Algorithm for structured diffing (and merging)
  - Think of `UNIX` diff, over algebraic datatypes.

- Wrote it in Haskell, generically

Contributions

- Representation for changes
- Efficient Algorithm for structured diffing (and merging)
  - Think of `UNIX` diff, over algebraic datatypes.
- Wrote it in Haskell, generically
- Evaluated against dataset from GitHub
  - mined Lua repositories

Line-by-Line Differencing

The UNIX diff

Compares files line-by-line, outputs an *edit script*.

```
type checker: "You fool!
What you request makes no sense,
rethink your bad code."
```

```
type checker: "You fool!
What you request makes no sense,
it's some ugly code."
```

The `UNIX` diff

Compares files line-by-line, outputs an *edit script*.

```
type checker: "You fool!
What you request makes no sense,
rethink your bad code."
```

```
type checker: "You fool!
What you request makes no sense,
it's some ugly code."
```

`UNIX` diff outputs:

```
@@ -3,1 , +3,1 @@
- rethink your bad code."
+ it's some ugly code."
```

The UNIX diff: In a Nutshell

Encodes changes as an *Edit Script*

```
data EOp        = Ins String | Del | Cpy


type EditScript = [EOp]
```

The UNIX diff: In a Nutshell

Encodes changes as an *Edit Script*

```
data EOp        = Ins String | Del | Cpy

type EditScript = [EOp]
```

Example,

```
@@ -3,1 , +3,1 @@
- rethink your bad code."
+ it's some ugly code."
```

```
[Cpy , Cpy , Del , Ins "it's some ..."]
```

The UNIX diff: In a Nutshell

Encodes changes as an *Edit Script*

```
data EOp        = Ins String | Del | Cpy


type EditScript = [EOp]
```

Example,

```
@@ -3,1 , +3,1 @@
- rethink your bad code."
+ it's some ugly code."
```

```
[Cpy , Cpy , Del , Ins "it's some ..."]
```

Computes changes by enumeration.

```
diff :: [String] -> [String] -> Patch
diff s d = head $ sortBy mostCopies $ enumerate_all s d
```

The UNIX diff: Abstractly

The UNIX diff: Abstractly

Abstractly, `diff` computes differences between two objects:

```
diff  :: a -> a -> Patch a
```

The UNIX diff: Abstractly

Abstractly, `diff` computes differences between two objects:

```
diff  :: a -> a -> Patch a
```

as a *transformation* that can be applied,

```
apply :: Patch a -> a -> Maybe a
```

The `UNIX` diff: Abstractly

Abstractly, `diff` computes differences between two objects:

```
diff  :: a -> a -> Patch a
```

as a *transformation* that can be applied,

```
apply :: Patch a -> a -> Maybe a
```

such that,

```
apply (diff s d) s == Just d
```

The `UNIX` diff: Abstractly

Abstractly, `diff` computes differences between two objects:

```
diff  :: a -> a -> Patch a
```

as a *transformation* that can be applied,

```
apply :: Patch a -> a -> Maybe a
```

such that,

```
apply (diff s d) s == Just d
```

`UNIX` diff works for `[String]`.

The UNIX diff Generalized: Edit Scripts

The UNIX diff Generalized: Edit Scripts

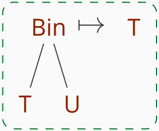
Modify Edit Scripts

```
data EOp = Ins TreeConstructor | Del | Cpy
```

The UNIX diff Generalized: Edit Scripts

Modify Edit Scripts

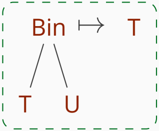
```
data EOp = Ins TreeConstructor | Del | Cpy
```



The UNIX diff Generalized: Edit Scripts

Modify Edit Scripts

```
data EOp = Ins TreeConstructor | Del | Cpy
```



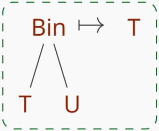
src tree preorder: [`Bin` , `T` , `U`]

dst tree preorder: [`T`]

The UNIX diff Generalized: Edit Scripts

Modify Edit Scripts

```
data EOp = Ins TreeConstructor | Del | Cpy
```



src tree preorder: `[Bin , T , U]`

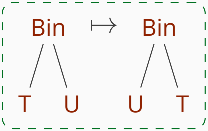
dst tree preorder: `[T]`

```
diff [Bin , T , U] [T] = [Del , Cpy , Del]
```

Edit Scripts: The Problem of Ambuiguity

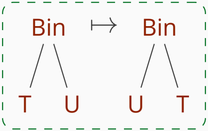
Edit Scripts: The Problem of Ambuiguity

Which subtree to copy?



Edit Scripts: The Problem of Ambuiguity

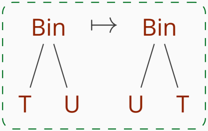
Which subtree to copy?



Copy `U` : [`Cpy , Del , Cpy , Ins T`]

Edit Scripts: The Problem of Ambuiguity

Which subtree to copy?

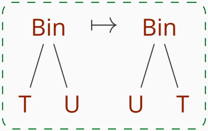


Copy `U` : [`Cpy` , `Del` , `Cpy` , `Ins T`]

Copy `T` : [`Cpy` , `Ins U` , `Cpy` , `Del`]

Edit Scripts: The Problem of Ambuiguity

Which subtree to copy?



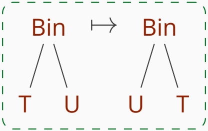
Copy `U` : [`Cpy , Del , Cpy , Ins T`]

Copy `T` : [`Cpy , Ins U , Cpy , Del`]

- Choice is **arbitrary**!

Edit Scripts: The Problem of Ambuiguity

Which subtree to copy?

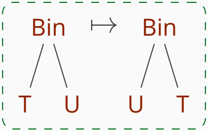


Copy `U` : [`Cpy` , `Del` , `Cpy` , `Ins T`]   Copy `T` : [`Cpy` , `Ins U` , `Cpy` , `Del`]

- Choice is **arbitrary**!
- Edit Script with the most copies is not unique!

Edit Scripts: The Problem of Ambuiguity

Which subtree to copy?



Copy `U` : [`Cpy` , `Del` , `Cpy` , `Ins T`]   Copy `T` : [`Cpy` , `Ins U` , `Cpy` , `Del`]

- Choice is **arbitrary**!
- Edit Script with the most copies is not unique!

Counting copies is reminiscent of longest common subsequence.

Edit Scripts: The Problem

Choice is necessary: only `Ins`, `Del` and `Cpy`

Edit Scripts: The Problem

Choice is necessary: only `Ins`, `Del` and `Cpy`

Drawbacks:

1. Cannot explore all copy oportunities: must chose one per subtree

Edit Scripts: The Problem

Choice is necessary: only `Ins`, `Del` and `Cpy`

Drawbacks:

1. Cannot explore all copy oportunities: must chose one per subtree

2. Choice points makes algorithms slow

Edit Scripts: The Problem

Choice is necessary: only `Ins`, `Del` and `Cpy`

Drawbacks:

1. Cannot explore all copy oportunities: must chose one per subtree

2. Choice points makes algorithms slow

Generalizations generalize specifications!

Edit Scripts: The Problem

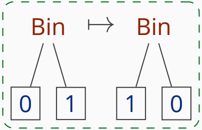
Choice is necessary: only `Ins`, `Del` and `Cpy`

Drawbacks:

1. Cannot explore all copy oportunities: must chose one per subtree
2. Choice points makes algorithms slow

Generalizations generalize specifications!

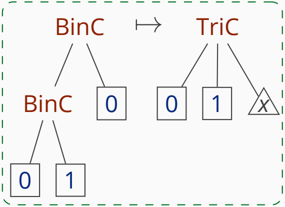
Solution: Detach from *edit-scripts*



New Structure for Changes

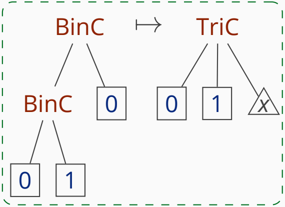
Changes

```
diff (Bin (Bin t u) t) (Tri t u x) =
```



Changes

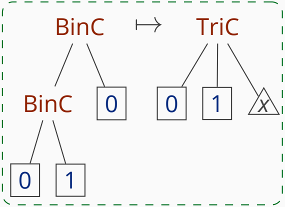
```
diff (Bin (Bin t u) t) (Tri t u x) =
```



- Arbitrary duplications, contractions, permutations
  - Can explore all copy opportunities

Changes

```
diff (Bin (Bin t u) t) (Tri t u x) =
```



- Arbitrary duplications, contractions, permutations
  - Can explore all copy opportunities
- Faster to compute
  - Our `diff s d` runs in $\mathcal{O}(\text{size s} + \text{size d})$

Changes

Two *contexts*

- deletion: matching
- insertion: instantiation

```
type Change = (TreeC MetaVar , TreeC MetaVar)


data Tree = Leaf
          | Bin Tree Tree
          | Tri Tree Tree Tree
```

Contexts are datatypes augmented with holes.

Changes

Two *contexts*

- deletion: matching
- insertion: instantiation

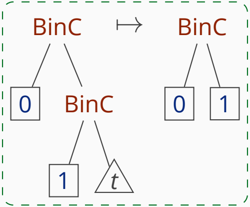
```
type Change = (TreeC MetaVar , TreeC MetaVar)
```

```
data Tree = Leaf
          | Bin Tree Tree
          | Tri Tree Tree Tree
```

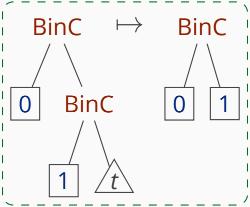
Contexts are datatypes augmented with holes.

```
data TreeC h = LeafC
             | BinC TreeC TreeC
             | TriC TreeC TreeC TreeC
             | Hole h
```

Applying Changes



Applying Changes



Application function sketch:

```
\x -> case x of
   Bin a (Bin b c) -> if c == t then Just (Bin a b) else Nothing
   _               -> Nothing
```

Computing Changes

Computing Changes

Can *copy as much as possible*

Computing Changes

Can *copy as much as possible*

Computation of `diff s d` can be split:

Computing Changes

Can *copy as much as possible*

Computation of `diff s d` can be split:

Hard Identify the common subtrees in `s` and `d`

Easy Extract the context around the common subtrees

Computing Changes

Can *copy as much as possible*

Computation of `diff s d` can be split:

Hard Identify the common subtrees in `s` and `d`

Easy Extract the context around the common subtrees

Consequence of definition of `Change`

Computing Changes

Can *copy as much as possible*

Computation of `diff s d` can be split:

Hard	Identify the common subtrees in `s` and `d`
Easy	Extract the context around the common subtrees

Consequence of definition of `Change`

Spec of the *hard* part:

```
wcs :: Tree -> Tree -> (Tree -> Maybe MetaVar)
wcs s d = flip elemIndex (subtrees s `intersect` subtrees d)
```

Computing Changes

Can *copy as much as possible*

Computation of `diff s d` can be split:

Hard Identify the common subtrees in `s` and `d`

Easy Extract the context around the common subtrees

Consequence of definition of `Change`

Spec of the *hard* part:

```
wcs :: Tree -> Tree -> (Tree -> Maybe MetaVar)
wcs s d = flip elemIndex (subtrees s `intersect` subtrees d)
```

Efficient `wcs` is akin to *hash-consing*. Runs in $\mathcal{O}(1)$.

Computing Changes: The Easy Part

Extracting the context:

```
extract :: (Tree -> Maybe MetaVar) -> Tree -> TreeC
extract f x = maybe (extract' x) Hole $ f x
  where
    extract' (Bin a b) = BinC (extract f a) (extract f b)
    ...
```

Computing Changes: The Easy Part

Extracting the context:

```
extract :: (Tree -> Maybe MetaVar) -> Tree -> TreeC
extract f x = maybe (extract' x) Hole $ f x
  where
    extract' (Bin a b) = BinC (extract f a) (extract f b)
    ...
```

Finally, with `wcs s d` as an *oracle*

```
diff :: Tree -> Tree -> Change MetaVar
diff s d = let o = wcs s d
            in (extract o s , extract o d)
```

Computing Changes: The Easy Part

Extracting the context:

```
extract :: (Tree -> Maybe MetaVar) -> Tree -> TreeC
extract f x = maybe (extract' x) Hole $ f x
  where
    extract' (Bin a b) = BinC (extract f a) (extract f b)
    ...
```

Finally, with `wcs s d` as an *oracle*

```
diff :: Tree -> Tree -> Change MetaVar
diff s d = let o = wcs s d
           in (extract o s , extract o d)
```

Since `wcs s d` is efficient, so is `diff s d`

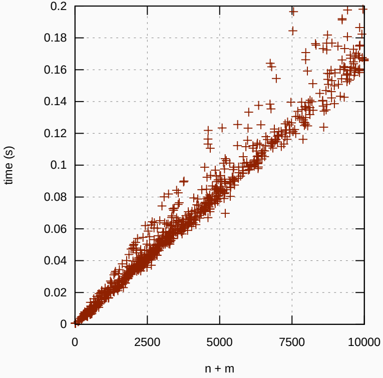
Experiments

Computing Changes: But how fast?

Diffed files from ≈ 1200 commits from top Lua repos

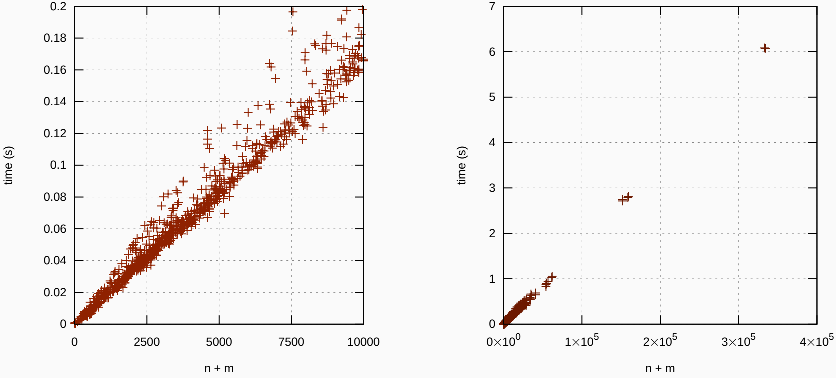
Computing Changes: But how fast?

Diffed files from ≈ 1200 commits from top Lua repos



Computing Changes: But how fast?

Diffed files from ≈ 1200 commits from top Lua repos



Merging Changes

Merging Changes

```
merge :: Change -> Change -> Either Conflict Change
merge p q = if p `disjoint` q then p else Conflict
```

Merging Changes

```haskell
merge :: Change -> Change -> Either Conflict Change
merge p q = if p `disjoint` q then p else Conflict
```

$$\begin{tikzcd}
 & s \arrow[dl, "p"'] \arrow[dr, "q"] & \\
d_1 \arrow[dr, "q"'] & & d_2 \arrow[dl, "p"] \\
 & r &
\end{tikzcd}$$

Merging Changes

```
merge :: Change -> Change -> Either Conflict Change
merge p q = if p `disjoint` q then p else Conflict
```

$$\begin{tikzcd}
& s \arrow[dl, "p"'] \arrow[dr, "q"] & \\
d_1 \arrow[dr, "q"'] & & d_2 \arrow[dl, "p"] \\
& r &
\end{tikzcd}$$

11% of all mined merge commits could be *automatically merged*

Open Questions

- How to reason over new change repr?

Open Questions

- How to reason over new change repr?
- Where do we stand with metatheory?

Open Questions

- How to reason over new change repr?
- Where do we stand with metatheory?
- Can’t copy bits inside a tree. Is this a problem?

Open Questions

- How to reason over new change repr?
- Where do we stand with metatheory?
- Can’t copy bits inside a tree. Is this a problem?
- ...

Summary

- Clear division of tasks (`wcs` oracle + context extraction)

Summary

- Clear division of tasks (`wcs` oracle + context extraction)
- Express more changes than edit scripts

Summary

- Clear division of tasks (`wcs` oracle + context extraction)
- Express more changes than edit scripts
- Faster algorithm than ES based tree-diff

Summary

- Clear division of tasks (`wcs` oracle + context extraction)
- Express more changes than edit scripts
- Faster algorithm than ES based tree-diff

- Overall:
  - Fast and generic algorithm
  - Encouraging empirical evidence

Utrecht University

Efficient Structural Differencing

Victor Cacciari Miraldo Wouter Swierstra

Utrecht University