Utrecht University

Efficient Structural Differencing

… and the lessons learned from it

Victor Cacciari Miraldo Wouter Swierstra

Utrecht University

Intro

Contributions

Contributions

- Efficient Algorithm for structured diffing (and merging)
    - Think of `UNIX` diff, over AST's.

Contributions

- Efficient Algorithm for structured diffing (and merging)
    - Think of `UNIX` diff, over AST's.
- Wrote it in Haskell, generically

Contributions

- Efficient Algorithm for structured diffing (and merging)
    - Think of `UNIX` diff, over AST's.
- Wrote it in Haskell, generically
- Tested 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 ES    = Ins String | Del | Cpy
type Patch = [ES]
```

The UNIX diff: In a Nutshell

Encodes changes as an *edit script*

```
data ES    = Ins String | Del | Cpy
type Patch = [ES]
```

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 ES    = Ins String | Del | Cpy
type Patch = [ES]
```

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 x y = head $ sortBy mostCopies $ enumerate_all x y
```

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 x y) x == Just y
```

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 x y) x == Just y
```

UNIX diff works for [`String`].

The UNIX diff Generalized: Edit Scripts

The UNIX diff Generalized: Edit Scripts

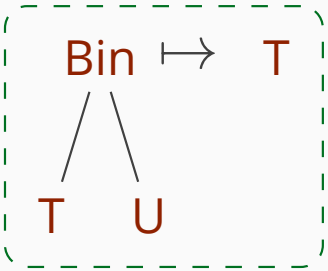
Modify edit scripts

```
data ES = Ins Tree | Del | Cpy
```

The UNIX diff Generalized: Edit Scripts

Modify edit scripts

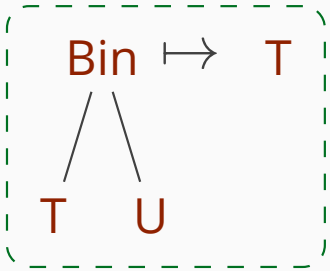
```
data ES = Ins Tree | Del | Cpy
```



The UNIX diff Generalized: Edit Scripts

Modify edit scripts

```
data ES = Ins Tree | Del | Cpy
```



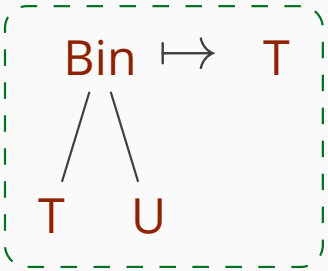
src tree preorder: `[Bin , T , U]`

dst tree preorder: `[T]`

The UNIX diff Generalized: Edit Scripts

Modify edit scripts

```
data ES = Ins Tree | Del | Cpy
```



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

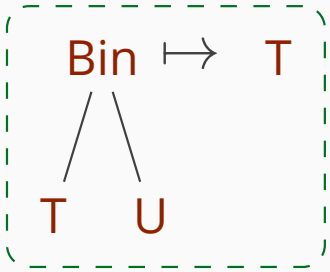
dst tree preorder: `[T]`

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

The UNIX diff Generalized: Edit Scripts

Modify edit scripts

```
data ES = Ins Tree | Del | Cpy
```



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

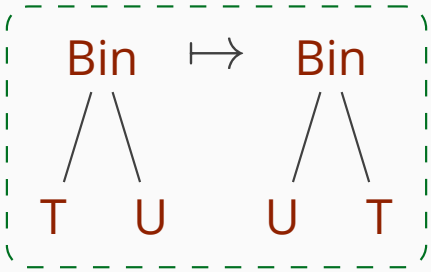
dst tree preorder: `[T]`

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

Not ideal

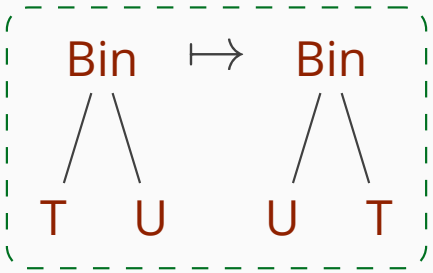
Edit Scripts: The Problem

Which subtree to copy?



Edit Scripts: The Problem

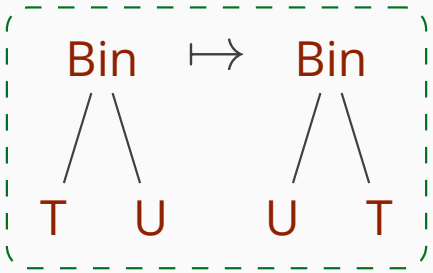
Which subtree to copy?



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

Edit Scripts: The Problem

Which subtree to copy?

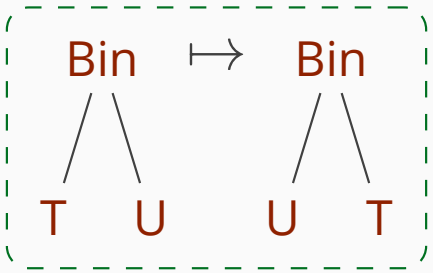


Copy `U` : `[Cpy , Del , Cpy , Ins T]`

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

Edit Scripts: The Problem

Which subtree to copy?



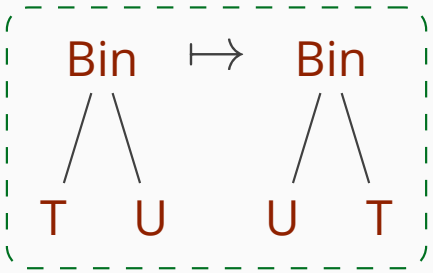
Copy `U` : `[Cpy , Del , Cpy , Ins T]`

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

- Choice is **arbitrary**!

Edit Scripts: The Problem

Which subtree to copy?



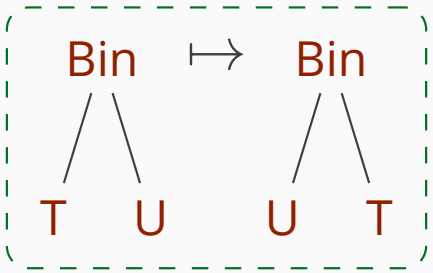
Copy `U` : `[Cpy , Del , Cpy , Ins T]`

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

- Choice is **arbitrary**!
- Counting Copies:
  - List case: corresponds to *longest common subseq.*

Edit Scripts: The Problem

Which subtree to copy?



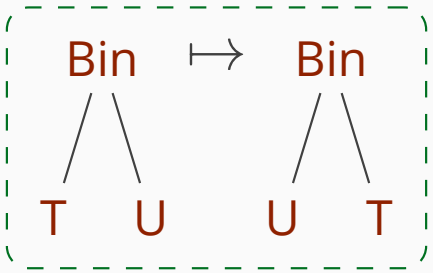
Copy `U` : `[Cpy , Del , Cpy , Ins T]`

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

- Choice is **arbitrary**!
- Counting Copies:
  - List case: corresponds to *longest common subseq.*
  - Tree case: Not so simple, most copies can be bad.

Edit Scripts: The Problem

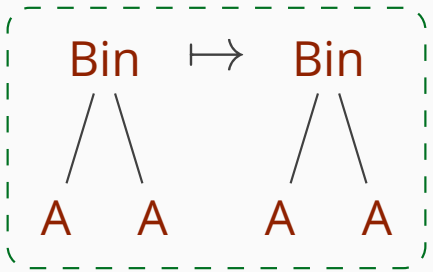
Which subtree to copy?



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

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

- Choice is **arbitrary**!
- Counting Copies:
  - List case: corresponds to *longest common subseq.*
  - Tree case: Not so simple, most copies can be bad.



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 can break specifications!

Edit Scripts: The Problem

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

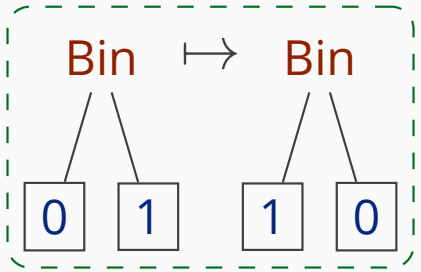
Drawbacks:

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

2. Choice points makes algorithms slow

Generalizations can break specifications!

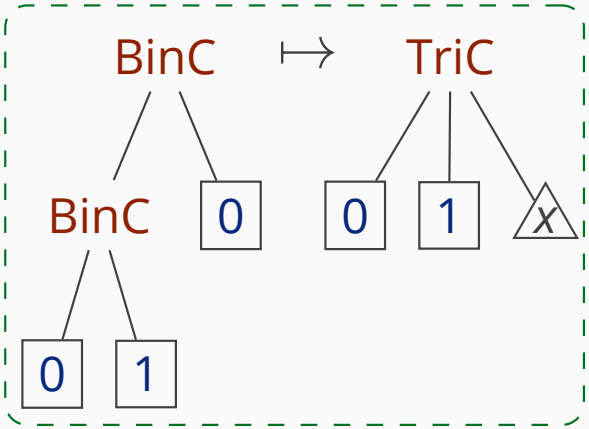
Solution: Detach from *edit-scripts*



New Structure for Changes

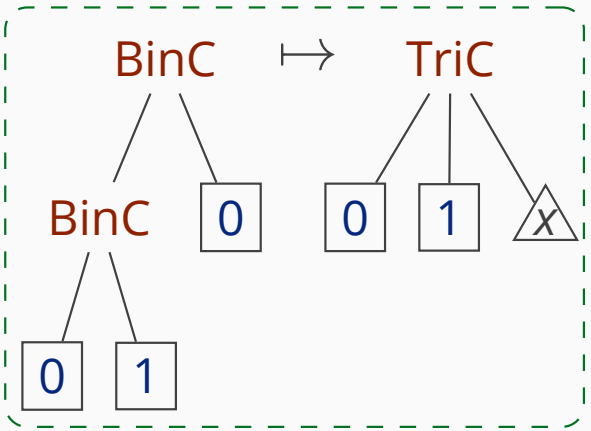
Changes

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



Changes

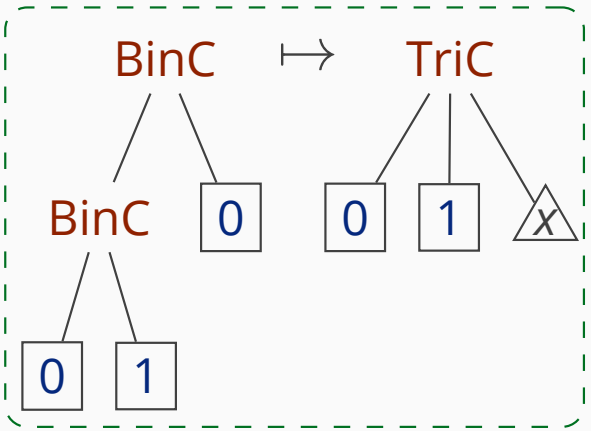
```
diff (Bin (Bin t u) t) (Tri t u x) =
```



- 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 x y` runs in $\mathcal{O}(\text{size x} + \text{size y})$

Changes

Two *contexts*

- deletion: matching
- insertion: instantiation

Changes

Two *contexts*

- deletion: matching
- insertion: instantiation

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

Context are datatypes annotated with holes.

Changes

Two *contexts*

- deletion: matching
- insertion: instantiation

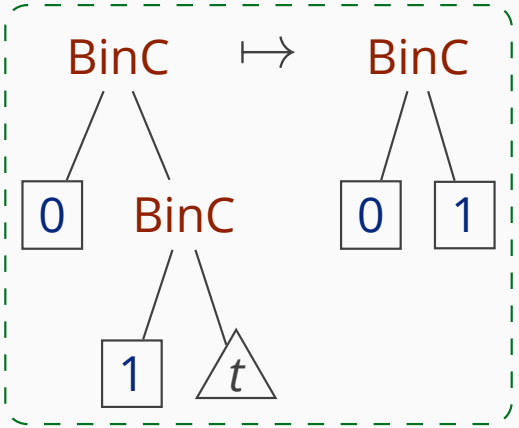
```
data Tree = Leaf
          | Bin Tree Tree
          | Tri Tree Tree Tree
```

Context are datatypes annotated with holes.

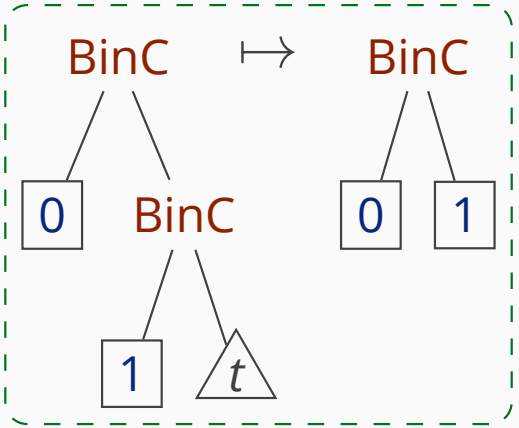
```
data TreeC h = LeafC
             | BinC TreeC TreeC
             | TriC TreeC TreeC TreeC
             | Hole h


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

Applying Changes

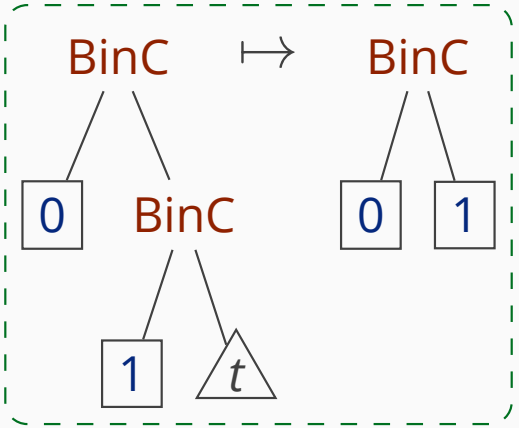


Applying Changes



Call it c,

Applying Changes



Call it `c`, application function sketch:

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

Relation to Edit Scripts

`Change` represents families of `ES`:

Relation to Edit Scripts

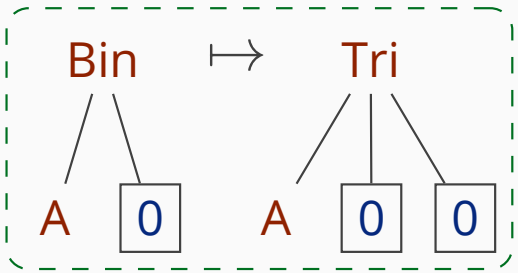
`Change` represents families of `ES`:

$$\texttt{Change} \approx \texttt{Tree} \to \texttt{Maybe}\ [\texttt{ES}]$$

Relation to Edit Scripts

`Change` represents families of `ES`:

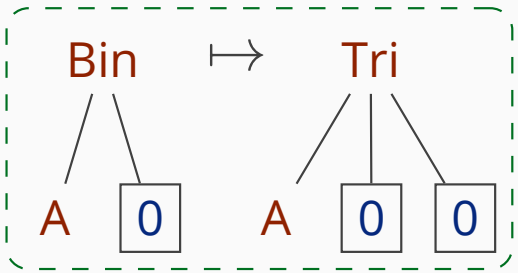
$$\texttt{Change} \approx \texttt{Tree} \to \texttt{Maybe}\ [\texttt{ES}]$$



Relation to Edit Scripts

`Change` represents families of `ES`:

$$\texttt{Change} \approx \texttt{Tree} \to \texttt{Maybe}\ [\texttt{ES}]$$



=

↦

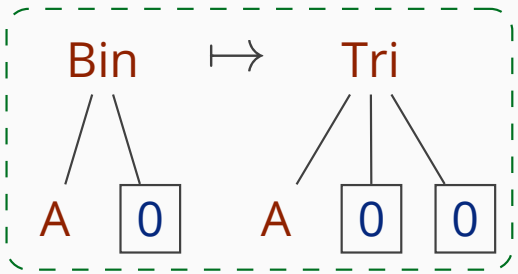
```
Just [
  [Del, Ins Tri, Cpy, Ins B, Cpy],
  [Del, Ins Tri, Cpy, Cpy, Ins B],
 ...]
```

↦

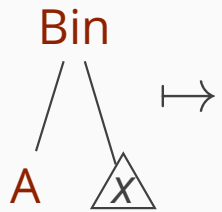
Relation to Edit Scripts

`Change` represents families of `ES`:

$$\texttt{Change} \approx \texttt{Tree} \to \texttt{Maybe [ES]}$$



$=$




```
↦  Just [
     [Del, Ins Tri, Cpy, Ins x, Cpy],
     [Del, Ins Tri, Cpy, Cpy, Ins x],
     ...]

_  ↦  Nothing
```

Computing Changes

Computing Changes

Can *copy as much as possible*

Computing Changes

Can *copy as much as possible*

Computation of `diff x y` divided:

Computing Changes

Can *copy as much as possible*

Computation of `diff x y` divided:

Hard Identify the common subtrees in `x` and `y`

Easy Extract the context around the common subtrees

Computing Changes

Can *copy as much as possible*

Computation of `diff x y` divided:

Hard Identify the common subtrees in `x` and `y`

Easy Extract the context around the common subtrees

Consequence of definition of `Change`

Computing Changes

Can *copy as much as possible*

Computation of `diff x y` divided:

Hard Identify the common subtrees in `x` and `y`

Easy Extract the context around the common subtrees

Consequence of definition of `Change`

Postpone the *hard* part for now

- Oracle: `wcs :: Tree -> Tree -> (Tree -> Maybe MetaVar)`
  - stands for *which common subtree*

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 = (extract (wcs s d) s , extract (wcs s d) 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 = (extract (wcs s d) s , extract (wcs s d) d)
```

if `wcs s d` is efficient, then so is `diff s d`

Computing Changes: Defining the Oracle

Computing Changes: Defining the Oracle

Defining an *inefficient* `wcs s d` is easy:

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

Computing Changes: Defining the Oracle

Defining an *inefficient* `wcs s d` is easy:

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

Efficient `wcs`:

- annotates `Tree` with cryptographic hashes, akin to a *Merkle Tree*
- store those in a `Trie` (amortized const. time search)
- uses topmost hash to compare trees for equality.

Computing Changes: Defining the Oracle

Defining an *inefficient* `wcs s d` is easy:

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

Efficient `wcs`:

- annotates `Tree` with cryptographic hashes, akin to a *Merkle Tree*
- store those in a `Trie` (amortized const. time search)
- uses topmost hash to compare trees for equality.

Runs in amortized $\mathcal{O}(1)$

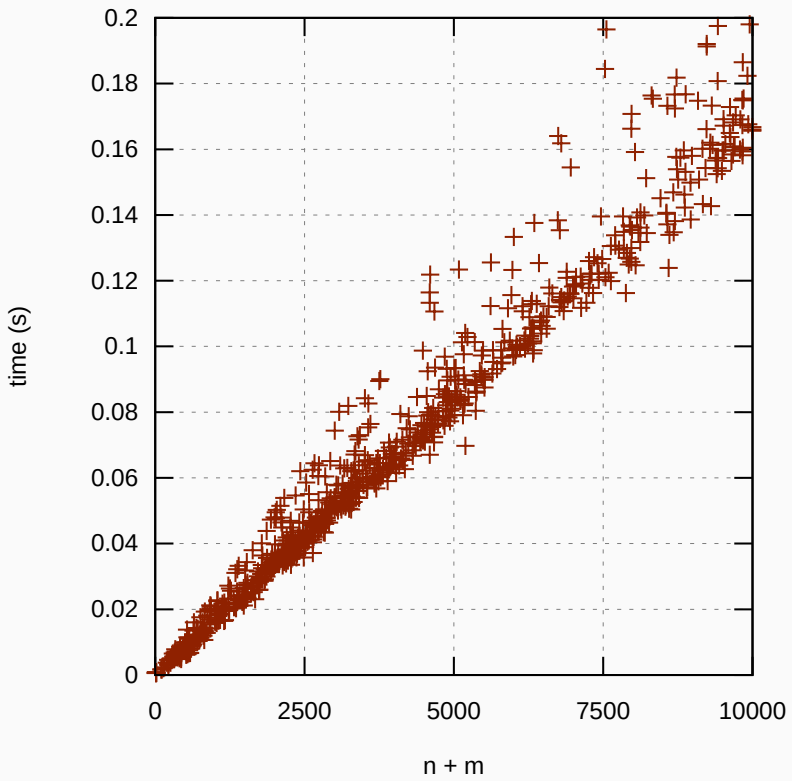
Experiments

Computing Changes: But how fast?

Diffed files from ≈ 1200 commits from top Lua repos

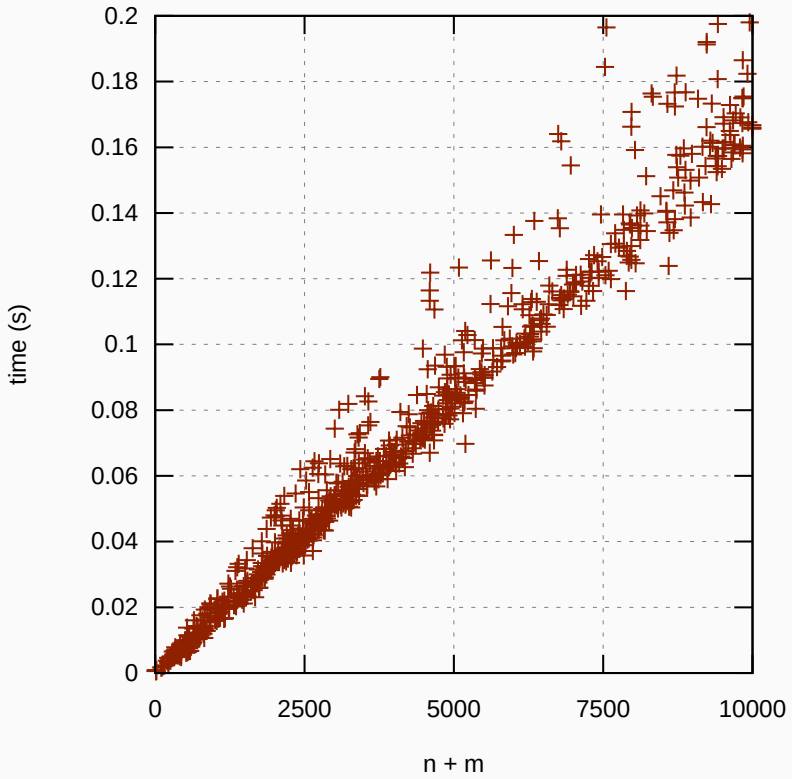
Computing Changes: But how fast?

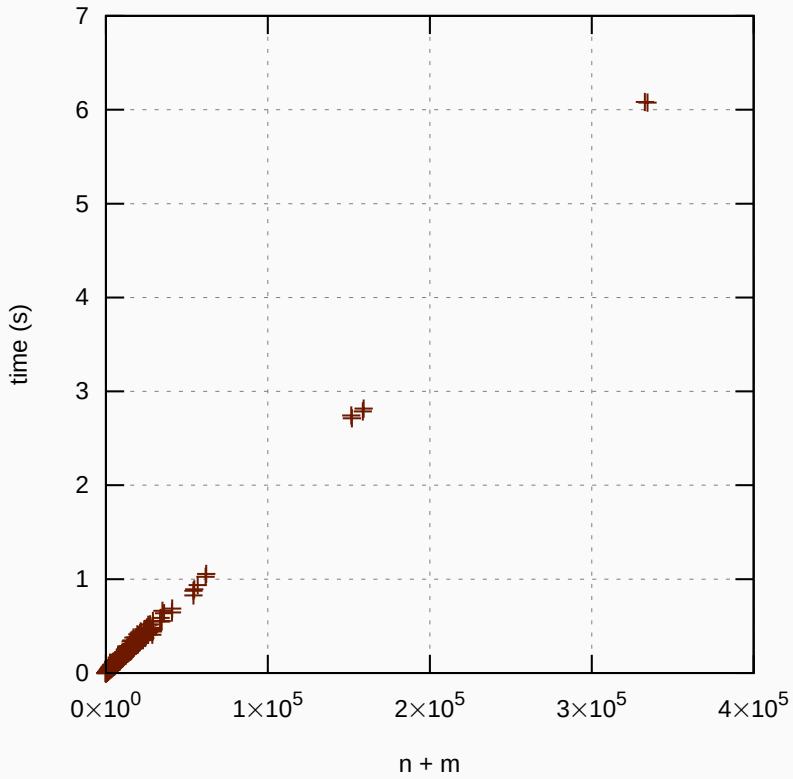
Diffed files from ≈ 1200 commits from top Lua repos



Computing Changes: But how fast?

Diffed files from ≈ 1200 commits from top Lua repos





Merging Changes

Merging is a constant motivation for structured diffing

Merging Changes

Merging is a constant motivation for structured diffing

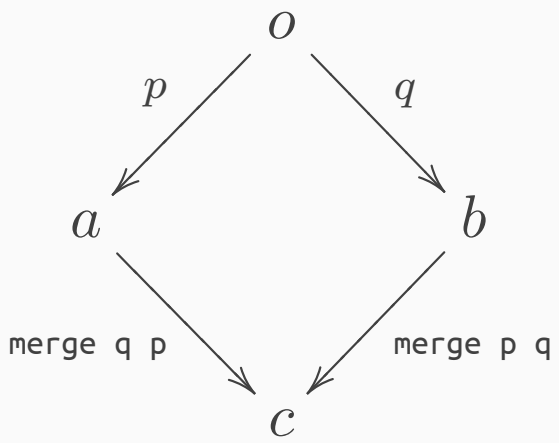
We defined a (very!) simple merging algorithm:

Merging Changes

Merging is a constant motivation for structured diffing

We defined a (very!) simple merging algorithm:

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

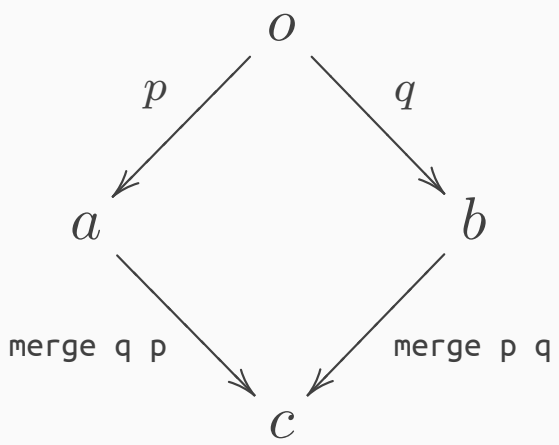


Merging Changes

Merging is a constant motivation for structured diffing

We defined a (very!) simple merging algorithm:

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



11% of mined merge commits could be *merged*

Summary

New representation enables:

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

Summary

New representation enables:

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

Summary

New representation enables:

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

Summary

New representation enables:

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

We have learned:

1. Generalizations can break specs

Summary

New representation enables:

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

We have learned:

1. Generalizations can break specs
2. More expressiveness does not mean higher complexity

Summary

New representation enables:

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

We have learned:

1. Generalizations can break specs
2. More expressiveness does not mean higher complexity
3. Thinking extensionally is very helpful

In Greater Depth

In Depth: The Efficient Oracle

Recall,

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

In Depth: The Efficient Oracle

Recall,

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

Two inefficiency points:

In Depth: The Efficient Oracle

Recall,

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

Two inefficiency points:

- Comparing trees for equality

In Depth: The Efficient Oracle

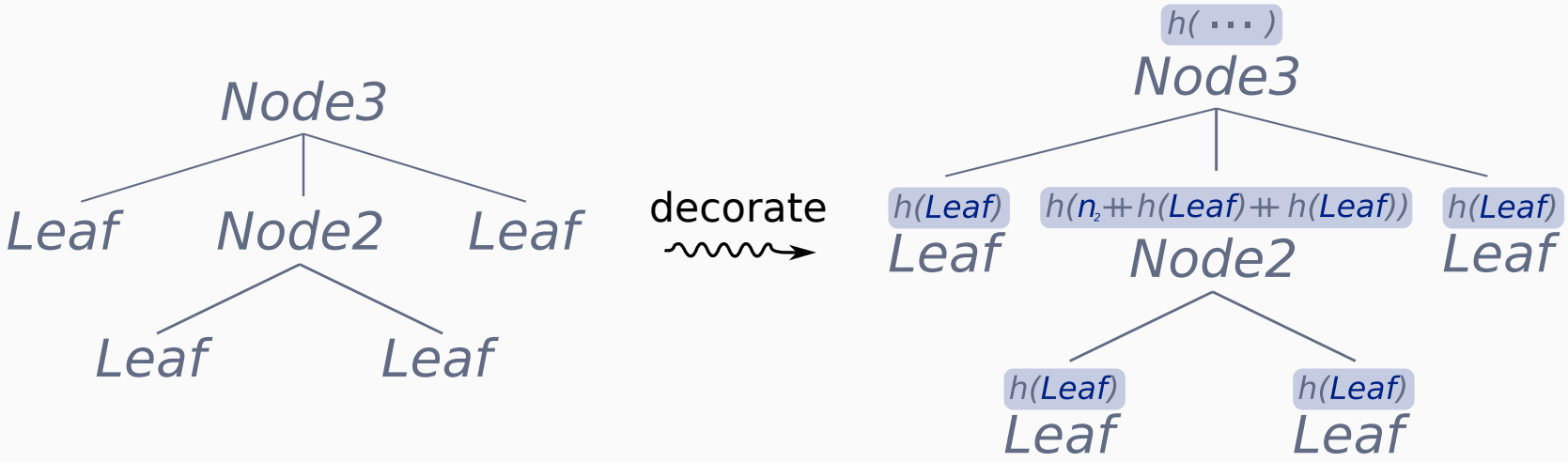
Recall,

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

Two inefficiency points:

- Comparing trees for equality
- Searching for a subtree in all enumerated subtrees

In Depth: The Efficient Oracle (Inefficiency #1)



In Depth: The Efficient Oracle (Merkle Trees)

In Depth: The Efficient Oracle (Merkle Trees)

Annotate Trees with `Digests`:

```
decorate :: Tree -> TreeH
data TreeH = LeafH
           | BinH (TreeH, Digest) (TreeH, Digest)
           | TriH (TreeH, Digest) (TreeH, Digest) (TreeH, Digest)
```

In Depth: The Efficient Oracle (Merkle Trees)

Annotate Trees with `Digests`:

```
decorate :: Tree -> TreeH
data TreeH = LeafH
           | BinH (TreeH, Digest) (TreeH, Digest)
           | TriH (TreeH, Digest) (TreeH, Digest) (TreeH, Digest)

root :: TreeH -> Digest
root LeafH                      = hash "leaf"
root (BinH (_ , dx) (_ , dy)) = hash ("node2" ++ dx ++ dy)
...
```

In Depth: The Efficient Oracle (Merkle Trees)

Annotate Trees with `Digests`:

```
decorate :: Tree -> TreeH
data TreeH = LeafH
           | BinH (TreeH, Digest) (TreeH, Digest)
           | TriH (TreeH, Digest) (TreeH, Digest) (TreeH, Digest)

root :: TreeH -> Digest
root LeafH                      = hash "leaf"
root (BinH (_ , dx) (_ , dy)) = hash ("node2" ++ dx ++ dy)
...
```

Compare roots:

```
instance Eq TreeH where
  t == u = root t == root u
```

In Depth: The Efficient Oracle (Inefficiency #2)

In Depth: The Efficient Oracle (Inefficiency #2)

Good structure to lookup hashes: **Tries**!

In Depth: The Efficient Oracle (Inefficiency #2)

Good structure to lookup hashes: **Tries**!

```
wcs :: TreeH -> TreeH -> (TreeH -> Maybe MetaVar)
wcs s d = lookup (tr empty s `intersect` tr empty d) . root
```

In Depth: The Efficient Oracle (Inefficiency #2)

Good structure to lookup hashes: **Tries**!

```
wcs :: TreeH -> TreeH -> (TreeH -> Maybe MetaVar)
wcs s d = lookup (tr empty s `intersect` tr empty d) . root

tr :: Trie -> TreeH -> Trie
tr db t = insert (root t)
        $ case t of
            LeafH                  -> db
            BinH (x , _) (y , _) -> tr (tr db x) y
            ...
```

In Depth: The Efficient Oracle: The `diff` function

One could write:

```
diff :: Tree -> Tree -> Change
diff s d = let s' = decorate s; d' = decorate d
           in (extract (wcs s' d') s' , extract (wcs s' d') d')
```

In Depth: The Efficient Oracle: The `diff` function

One could write:

```
diff :: Tree -> Tree -> Change
diff s d = let s' = decorate s; d' = decorate d
           in (extract (wcs s' d') s' , extract (wcs s' d') d')
```

Subtle issue: `a = Bin (Bin t k) u; b = Bin (Bin t k) t`

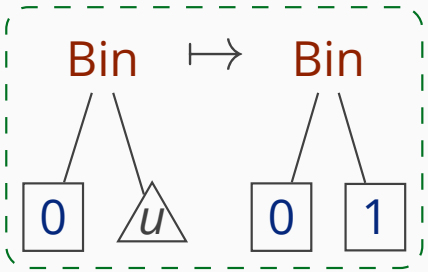
In Depth: The Efficient Oracle: The `diff` function

One could write:

```
diff :: Tree -> Tree -> Change
diff s d = let s' = decorate s; d' = decorate d
           in (extract (wcs s' d') s' , extract (wcs s' d') d')
```

Subtle issue: `a = Bin (Bin t k) u; b = Bin (Bin t k) t`

Wrong



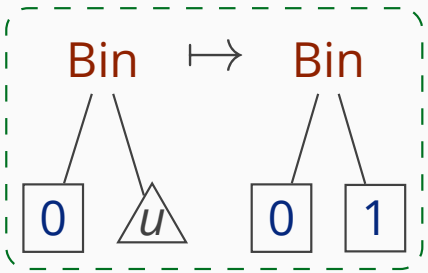
In Depth: The Efficient Oracle: The `diff` function

One could write:

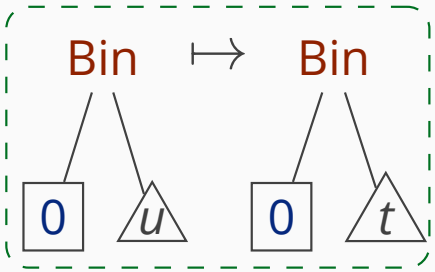
```
diff :: Tree -> Tree -> Change
diff s d = let s' = decorate s; d' = decorate d
           in (extract (wcs s' d') s' , extract (wcs s' d') d')
```

Subtle issue: `a = Bin (Bin t k) u; b = Bin (Bin t k) t`

Wrong



Correct:



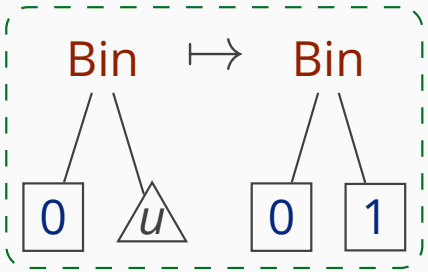
In Depth: The Efficient Oracle: The `diff` function

One could write:

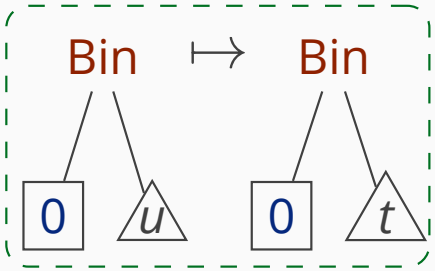
```
diff :: Tree -> Tree -> Change
diff s d = let s' = decorate s; d' = decorate d
           in (extract (wcs s' d') s' , extract (wcs s' d') d')
```

Subtle issue: `a = Bin (Bin t k) u`; `b = Bin (Bin t k) t`

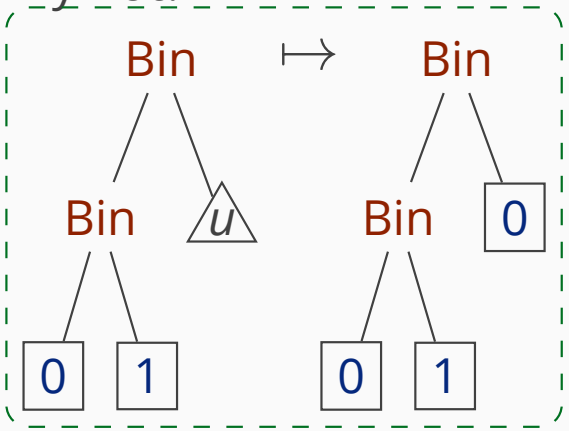
Wrong



Correct:



Why not?



In Depth: The “best” change

In Depth: The “best” change

- The “best” change is the one with the largest domain.
- least specific

In Depth: The “best” change

- The “best” change is the one with the largest domain.
- least specific

Let c and d be changes that transform x into y.

In Depth: The “best” change

- The “best” change is the one with the largest domain.
- least specific

Let c and d be changes that transform x into y.

$$c \subseteq d \Leftrightarrow \exists \sigma \ . \ \mathrm{dom}\ c \sqsubseteq_{\sigma} \mathrm{dom}\ d$$

In Depth: The “best” change

- The “best” change is the one with the largest domain.
- least specific

Let c and d be changes that transform x into y.

$$c \subseteq d \Leftrightarrow \exists \sigma \ . \ \mathrm{dom}\ c \sqsubseteq_\sigma \mathrm{dom}\ d$$

$$\frac{}{x \sqsubseteq_\sigma x} \qquad \frac{t = \sigma\, x}{x \sqsubseteq_\sigma t} \qquad \frac{x_1 \sqsubseteq_\sigma y_1 \qquad x_2 \sqsubseteq_\sigma y_2 \qquad \cdots}{\mathrm{C}\,\vec{x} \sqsubseteq_\sigma \mathrm{C}\,\vec{y}}$$

In Depth: The “best” change

- The “best” change is the one with the largest domain.
- least specific

Let c and d be changes that transform x into y.

$$c \subseteq d \Leftrightarrow \exists \sigma \; . \; \mathrm{dom}\; c \sqsubseteq_\sigma \mathrm{dom}\; d$$

$$\frac{}{x \sqsubseteq_\sigma x} \qquad \frac{t = \sigma\, x}{x \sqsubseteq_\sigma t} \qquad \frac{x_1 \sqsubseteq_\sigma y_1 \qquad x_2 \sqsubseteq_\sigma y_2 \qquad \cdots}{\mathrm{C}\,\vec{x} \sqsubseteq_\sigma \mathrm{C}\,\vec{y}}$$

This makes a preorder (reflexive; transitive)

In Depth: Merging

Hard to reason with `Change`

In Depth: Merging

Hard to reason with `Change`

- Redundant Info
- Metavariable Scope

In Depth: Merging

Hard to reason with `Change`

- Redundant Info
- Metavariable Scope

un-*distribute* the redundant constructors.

```
type Patch = TreeC Change
```

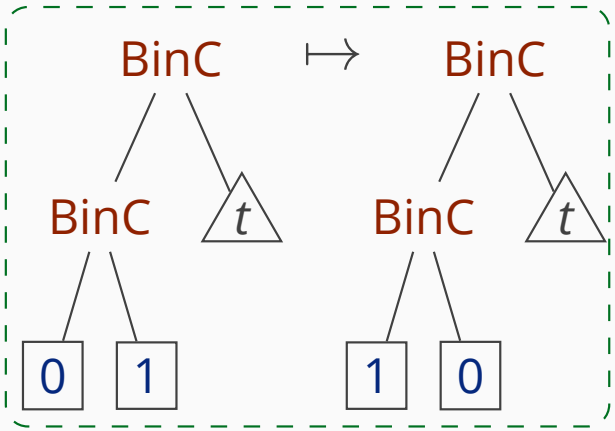
In Depth: Merging

Hard to reason with `Change`

- Redundant Info
- Metavariable Scope

un-*distribute* the redundant constructors.

```
type Patch = TreeC Change
```



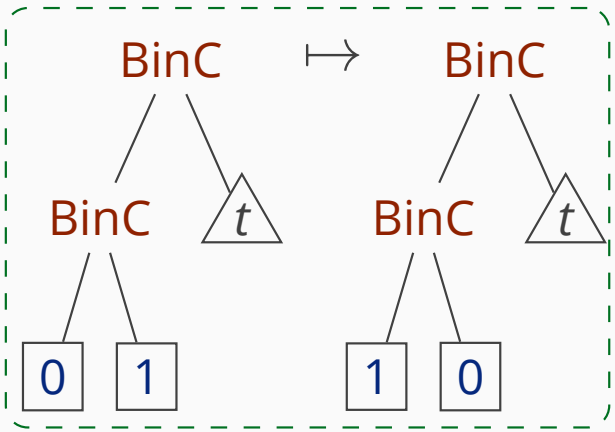
In Depth: Merging

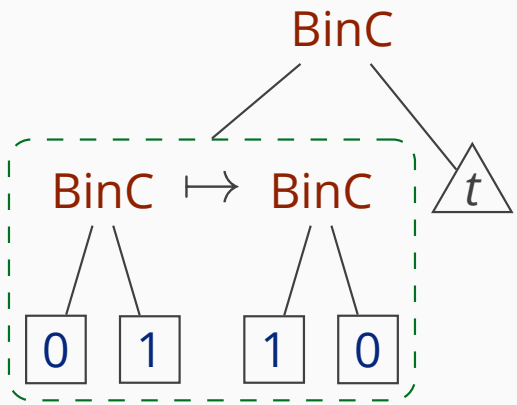
Hard to reason with `Change`

- Redundant Info
- Metavariable Scope

un-*distribute* the redundant constructors.

```
type Patch = TreeC Change
```





In Depth: Merging and Anti-unification

In Depth: Merging and Anti-unification

Extract the *greatest common prefix* from two `TreeC`:

```
gcp :: TreeC a -> TreeC b -> TreeC (TreeC a , TreeC b)
gcp LeafC         LeafC         = LeafC
gcp (BinC x y)    (BinC u v)    = BinC (gcp x u) (gcp y v)
gcp (TriC x y z) (TriC u v w) = TriC (gcp x u) (gcp y v) (gcp z w)
gcp x             w             = Hole (x , y)
```

In Depth: Merging and Anti-unification

Extract the *greatest common prefix* from two `TreeC`:

```
gcp :: TreeC a -> TreeC b -> TreeC (TreeC a , TreeC b)
gcp LeafC         LeafC         = LeafC
gcp (BinC x y)    (BinC u v)    = BinC (gcp x u) (gcp y v)
gcp (TriC x y z) (TriC u v w) = TriC (gcp x u) (gcp y v) (gcp z w)
gcp x             w             = Hole (x , y)
```

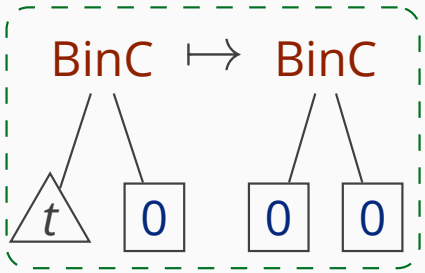
Problematic. Can break scoping.

In Depth: Merging and Anti-unification

Extract the *greatest common prefix* from two `TreeC`:

```
gcp :: TreeC a -> TreeC b -> TreeC (TreeC a , TreeC b)
gcp LeafC         LeafC         = LeafC
gcp (BinC x y)    (BinC u v)    = BinC (gcp x u) (gcp y v)
gcp (TriC x y z) (TriC u v w) = TriC (gcp x u) (gcp y v) (gcp z w)
gcp x             w             = Hole (x , y)
```

Problematic. Can break scoping.

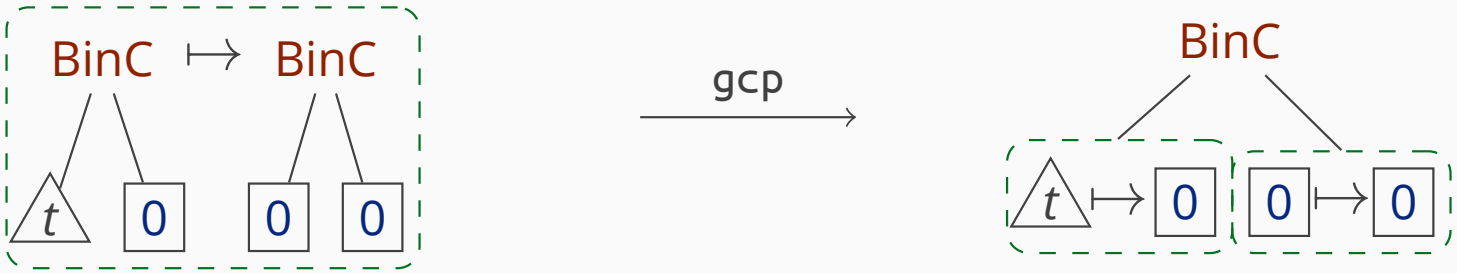


In Depth: Merging and Anti-unification

Extract the *greatest common prefix* from two `TreeC`:

```
gcp :: TreeC a -> TreeC b -> TreeC (TreeC a , TreeC b)
gcp LeafC         LeafC         = LeafC
gcp (BinC x y)    (BinC u v)    = BinC (gcp x u) (gcp y v)
gcp (TriC x y z) (TriC u v w) = TriC (gcp x u) (gcp y v) (gcp z w)
gcp x             w             = Hole (x , y)
```

Problematic. Can break scoping.

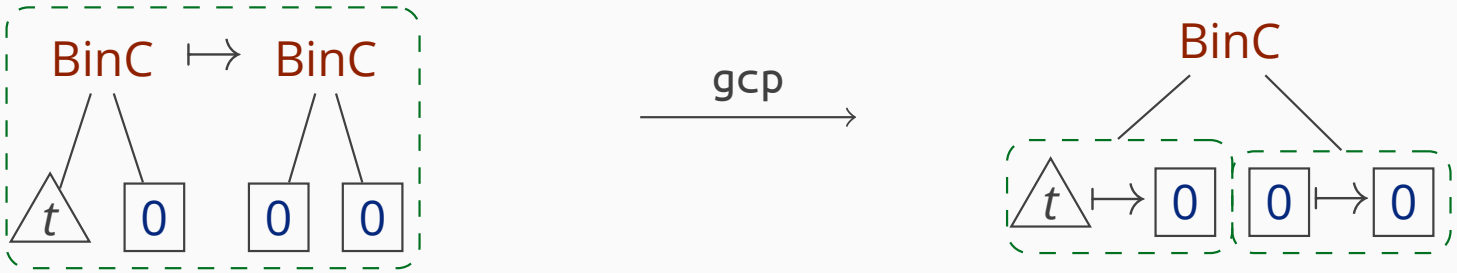


In Depth: Merging and Anti-unification

Extract the *greatest common prefix* from two `TreeC`:

```
gcp :: TreeC a -> TreeC b -> TreeC (TreeC a , TreeC b)
gcp LeafC         LeafC         = LeafC
gcp (BinC x y)    (BinC u v)    = BinC (gcp x u) (gcp y v)
gcp (TriC x y z) (TriC u v w) = TriC (gcp x u) (gcp y v) (gcp z w)
gcp x             w             = Hole (x , y)
```

Problematic. Can break scoping.



Define `closure :: Patch -> Patch` to fix scopes.

Discussion

Discussion

Performance of structural diffing:

Discussion

Performance of structural diffing: Fixed

Discussion

Performance of structural diffing: Fixed

Now what?

- Metatheory

Discussion

Performance of structural diffing: Fixed

Now what?

- Metatheory
- Sharing Control

Discussion

Performance of structural diffing: Fixed

Now what?

- Metatheory
- Sharing Control
- Merge Strategies

Discussion

Performance of structural diffing: Fixed

Now what?

- Metatheory
- Sharing Control
- Merge Strategies
- Domain-specific conflict resolution

Discussion

Performance of structural diffing: Fixed

Now what?

- Metatheory
- Sharing Control
- Merge Strategies
- Domain-specific conflict resolution
- Bigger univeses

Discussion

Performance of structural diffing: Fixed

Now what?

- Metatheory
- Sharing Control
- Merge Strategies
- Domain-specific conflict resolution
- Bigger univeses

Utrecht University

Efficient Structural Differencing

… and the lessons learned from it

Victor Cacciari Miraldo Wouter Swierstra

Utrecht University