Dating tree rings: CST-based version control

Gustek
gustek@riseup.net

Abstract

1. Introduction

The use of version control systems (VCS) is ubiquitous in the software development industry. The core of a VCS can be identified as two main processes: the calculation of changes between two versions of the program, and the merging of the said changes when they exist across different branches. Most VCSs — such as Git (Torvalds 2005), which is almost universally used in open source software projects — perform this first step in a similar fashion as `diff(1)`; that is, linearly. This strategy, although simple to implement, is unsatisfactory and suboptimal. This arises from three problems, two of which happen to be the very problems VCSs have to solve. Consider the following versions of a file:

```
...
really_long_function_name(5);
...
```

```
...
really_long_function_name(6);
...
```

Using a linear diff algorithm, the whole line is considered changed even if only a single character of the line has actually been modified.

Let us now consider Figure 1. Version `a` is the "base" version of the file, whereas version `b` and `c` each succeed it on a different branch.

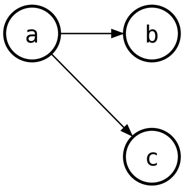

Figure 1: Position of `a`, `b` and `c` in the history

version `a`:

```
...
5 + 6
...
```

version `b`:

```
...
5 + 7
...
```

version `c`:

```
...
5 - 6
...
```

If we try and merge version `b` and `c` while the changes they describe have been calculated linearly, a merge conflict will occur, as the same line has been changed in two different ways, although there is no real conflict on a syntactical level. Such conflicts, especially when multiplied — as they tend to be — are very time-cosuming to fix and greatly impair productivity, requiring human intervention on a task that should be peformed automatically.

The third problem line-based VCS (or diff programs in general) exhibit is the lack of clarity for the user. See the following example:

version 1:

version 2:

```
fn f(a: i32, x: i32) -> i32 {
  (if x % 2 == 0 {
    -1
  } else {
    x
  }) + a
}

fn main() {
  let v = vec![1, 2, 3];
  let x = v.iter().fold(0, f);
}
```

```
fn f(a: i32, x: i32) -> i32 {
  (if x % 2 == 0 {
    2
  } else {
    x
  }) + a
}

fn main() {
  let v = vec![1, 2, 3];
  let x = v.iter().fold(0, f);
}
```

The difference between both versions as calculated by a linear algorithm is the replacement of the line containing `-1` by one containing `2`. It would be difficult for the user to figure out what the change represents and he couldn't have more information on the actual nature of the change, given that the linear diff is not syntax-aware.

In this paper, we study the computation of *diffs* (ie. collections of changes between two versions of a program) of arborescent structures and the merging of such diffs. By applying such computations to syntax trees, the problems highlighted in the previous examples would be solved, as the difference between the two lines of code of the first example would be reduced to $5 \longrightarrow 6$ (resulting in smaller diff files), there would be no merge conflicts in the history described by Figure 1, given that version `b` modifies an *operand* whereas version `c` changes an *operator*, and syntax-awareness would allow for helpful contextualisation when displaying diff files to the user.

In this article, we tackle the issue of producing an optimal diff for recursive structures. For doing so, we introduce an expressive language for representing structural changes and present algorithms for calculating changes and applying them to recursive structures. We prove the correction of these algorithms and discuss both theoretical and practical optimisations. We also bring forth an algorithm for merging structural changes, proving the correction thereof. Finally, we compare the performance of our solution to linear diffs and existing structural analysers in real-world situations and review the existing literature and implementations on this topic.

2. Diffs and trees

The algorithms we describe here are process binary trees T defined as follows:

$$\begin{aligned} \mathrm{T} ::= {} & \kappa : A \longrightarrow \mathrm{T} \\ \mid {} & \tau_i : \mathrm{T} \longrightarrow \mathrm{T} \longrightarrow \mathrm{T} \text{ where } i : B \end{aligned}$$

The types A and B respectively represent a "data" type and a "metadata" type for the trees. The only constraint placed upon them is that there exists an equivalence relation for each of them.

However, most parsers return the children of nodes as a *list* of trees (ie. concrete syntax trees as rose trees). We thus define a conversion function from such a tree (written T_R) to a binary tree T and backwards. We also define two utilitary values: cons_B, the metadata marker for a converted cons cell and $\mathrm{nil}_{\mathrm{T}}$, a special variant of κ. In describing the conversion algorithm, we use linked list with the usual `cons` and `nil` functions. Let $c_{r\to b} : \mathrm{T}_R \longrightarrow \mathrm{T}$ and $c_{b\to r} : \mathrm{T} \longrightarrow \mathrm{T}_R$ respectively be the conversion function from rose trees to binary trees and vice-versa:

$$\begin{aligned} c_{r\to b}(\kappa_R(x)) &= \kappa(x) \\ c_{r\to b}(\tau_{Ri}(\mathrm{cons}(x, \mathrm{nil}))) &= \tau_i(c_{r\to b}(x), \mathrm{nil}_{\mathrm{T}}) \\ c_{r\to b}(\tau_{Ri}(\mathrm{cons}(x, x'))) &= \tau_i\left(c_{r\to b}(x), c_{r\to b}\left(\tau_{R\mathrm{cons}_B}(x')\right)\right) \end{aligned}$$

$$c_{r\to b}(\tau_{Ri}(\text{nil})) = \tau_i(\text{nil}_{\text{T}}, \text{nil}_{\text{T}})$$

$$\begin{aligned} c_{b\to r}(\kappa(x)) &= \kappa_R(x) \\ c_{b\to r}(\tau_i(x, \text{nil}_{\text{T}})) &= \tau_{Ri}(\text{cons}(c_{b\to r}(x), \text{nil})) \\ c_{b\to r}(\tau_i(x, y)) &= \tau_{Ri}(c_{b\to r}(x) :: {c_{b\to r}}'(y)) \end{aligned}$$

where ${c_{b\to r}}' : \text{T} \longrightarrow \text{list } \text{T}_R$ is a utilitary function that is defined as follows:

$$\begin{aligned} {c_{b\to r}}'(\text{nil}_T) &= \text{nil} \\ {c_{b\to r}}'\left(\tau_{\text{cons}_B}(x, y)\right) &= \text{cons}(c_{b\to r}(x), {c_{b\to r}}'(y)) \end{aligned}$$

All the cases that are unmatched by the $c_{b\to r}$ (and incidentally ${c_{b\to r}}'$) function correspond to badly-formed binary trees and should return an error when encountered

Lemma 2.1 (Conversion correctness): $\forall t : \text{T}_R, c_{b\to r}(c_{r\to b}(t)) = t$

Proof: See Appendix A.1 Q.E.D.

We can now define a diff type Δ to represent changes between binary trees. It can be seen that its structure is much more complex than that of unidimensional (ie. linear) diffs.

$$\begin{aligned} \Delta ::= \; & \varepsilon : \Delta \\ & \mid t_{\varepsilon i} : \Delta \longrightarrow \Delta \longrightarrow \Delta \\ & \mid \mu : \text{T} \longrightarrow \text{T} \longrightarrow \Delta \\ & \mid t_{\mu i \to j} : \Delta \longrightarrow \Delta \longrightarrow \Delta \\ & \mid \pi_{\dashv i} : \text{T} \longrightarrow \Delta \longrightarrow \Delta \\ & \mid \pi_{\vdash i} : \Delta \longrightarrow \text{T} \longrightarrow \Delta \\ & \mid \beta_{\dashv} : \Delta \longrightarrow \Delta \\ & \mid \beta_{\vdash} : \Delta \longrightarrow \Delta \end{aligned}$$

ε indicates the absence of change between two binary trees. t_ε indicates an equality in node type (and thus that the computation of changes follows on the next level). μ formalises the *modification* of a node, while t_μ signifies the modfication of the node *type* between the left and right trees (and indicates the lower-level changes). $\pi_\dashv$ and $\pi_\vdash$ indicate the addition of a depth level, defining an arbitrary tree as the respectively left and right child of the new node and indicating the calculated changes for the new node's (respectively) right and left child. Conversely, $\beta_\dashv$ and $\beta_\vdash$ indicate the deletion of a node and the continuation of the computation on the right and the left, respectively, discarding the other-hand child.

We define a weight function $w : \Delta \longrightarrow \mathbb{N}$ on diffs, indicative of the cost of applying (and storing) the diff (nb. $|x|$ is the size of $x : \text{T}$).

$$\begin{aligned} w(\varepsilon) &= 0 \\ w(t_{\varepsilon i}(x, y)) &= w(x) + w(y) \\ w(\mu(x, y)) &= 1 + |x| + |y| \\ w\left(t_{\mu i \to j}(x, y)\right) &= 1 + w(x) + w(y) \end{aligned}$$

$$\begin{aligned} w\Big(\pi_{\dashv/\vdash i}(t,\delta)\Big) &= 1 + |t| + w(\delta) \\ w\Big(\beta_{\dashv/\vdash i}(\delta)\Big) &= 1 + w(\delta) \end{aligned}$$

We also define a $\min_w : \Delta \longrightarrow \Delta \longrightarrow \Delta$ function, yielding the diff having the smallest weight of the two, along with its generalisation for every $n \in \mathbb{N}^*$, $\min_w : \Delta^n \longrightarrow \Delta$.

3. Diffing and patching

3.1. Principle

If we represent trees and diffs as an arithmetical system, we can define the diff operation as an external substraction $- : \mathrm{T} \longrightarrow \mathrm{T} \longrightarrow \Delta$, such that $\delta = y - x$. We can then define the patch operation as an external addition $+ : \mathrm{T} \longrightarrow \Delta \longrightarrow \mathrm{T}$, such that $x + \delta = y$. It then follows that $x + (y - x) = y$. The diff function can be described as “ε-potent”, given that $x - x = \varepsilon$.

It is worth noting that the patch function is not actually defined on $\mathrm{T} \longrightarrow \Delta \longrightarrow \mathrm{T}$, rather on $\mathrm{T} \longrightarrow \Delta_t \longrightarrow \mathrm{T}$, where Δ_t is the set of diffs applicable to a specific tree t, on which we can place the following bound: $\{\varepsilon\} \subset \Delta_t$.

3.2. Algorithms

We thus define the diff function $d : \mathrm{T} \longrightarrow \mathrm{T} \longrightarrow \Delta$:

$$d(\kappa(x), \kappa(y)) = \begin{cases} \varepsilon \text{ if } x = y \\ \mu(x, y) \text{ else} \end{cases}$$

$$\begin{aligned} d\big(\tau_i(x,y), \tau_j(x',y')\big) = &\begin{cases} \min_w\Big(\delta_\varepsilon, \delta_{\pi_\dashv}, \delta_{\pi_\vdash}, \delta_{\beta_\dashv}, \delta_{\beta_\vdash}\Big) \text{ if } i = j \\ \min_w\Big(\delta_\mu, \delta_{t\mu}, \delta_{\pi_\dashv}, \delta_{\pi_\vdash}, \delta_{\beta_\dashv}, \delta_{\beta_\vdash}\Big) \text{ else} \end{cases} \\ \text{where } \delta_\varepsilon &= t_{\varepsilon i}(d(x,x'), d(y,y')) \\ \delta_{t\mu} &= t_{\mu i \to j}(d(x,x'), d(y,y')) \\ \delta_\mu &= \mu\big(\tau_i(x,y), \tau_j(x',y')\big) \\ \delta_{\pi_\dashv} &= \pi_{\dashv j}(x', d(\tau_i(x,y), y')) \\ \delta_{\pi_\vdash} &= \pi_{\vdash j}(d(\tau_i(x,y), x'), y') \\ \delta_{\beta_\dashv} &= \beta_\dashv\big(d\big(y, \tau_j(x',y')\big)\big) \\ \text{and } \delta_{\beta_\vdash} &= \beta_\vdash\big(d\big(x, \tau_j(x',y')\big)\big) \end{aligned}$$

$$\begin{aligned} d(\kappa(a), \tau_i(x,y)) &= \min_w\Big(\delta_\mu, \delta_{\pi_\dashv}, \delta_{\pi_\vdash}\Big) \\ \text{where } \delta_\mu &= \mu(\kappa(a), \tau_i(x,y)) \\ \delta_{\pi_\dashv} &= \pi_{\dashv i}(x, d(\kappa(a), y)) \\ \text{and } \delta_{\pi_\vdash} &= \pi_{\vdash i}(y, d(\kappa(a), x)) \end{aligned}$$

$$\begin{aligned} d(\tau_i(x,y), \kappa(a)) &= \min_w\Big(\delta_\mu, \delta_{\beta_\dashv}, \delta_{\beta_\vdash}\Big) \\ \text{where } \delta_\mu &= \mu(\tau_i(x,y), \kappa(a)) \\ \delta_{\beta_\dashv} &= \beta_\dashv(d(y, \kappa(a))) \\ \text{and } \delta_{\beta_\vdash} &= \beta_\vdash(d(x, \kappa(a)) \end{aligned}$$

We then define the patch function $p : \mathrm{T} \longrightarrow \Delta \longrightarrow \mathrm{T}$:

$$\begin{aligned}
p(x, \varepsilon) &= x \\
p(x, \mu(x, y)) &= y \\
p(\tau_i(x, y), t_{\varepsilon i}(\delta_x, \delta_y)) &= \tau_i(p(x, \delta_x), p(y, \delta_y)) \\
p(x, \pi_{\dashv i}(x', \delta_y)) &= \tau_i(x', p(x, \delta_y)) \\
p(x, \pi_{\vdash i}(y', \delta_x)) &= \tau_i(p(x, \delta_x), y') \\
p(\tau_i(\_, y), \beta_{\dashv}(\delta_y)) &= p(y, \delta_y) \\
p(\tau_i(x, \_), \beta_{\vdash}(\delta_x)) &= p(x, \delta_x) \\
p(\tau_i(x, y), t_{\mu i \to j}(\delta_x, \delta_y)) &= \tau_j(p(x, \delta_x), p(y, \delta_y))
\end{aligned}$$

One can see that the definition of p does not match the entirety of $\mathrm{T} \times \Delta$. In such cases not defined here, an implementation of the algorithm should throw an error, indicating that the provided diff is incompatible with the tree.

3.3. Correctness

In this section, we shall prove the correctness of the diff-patch pipeline. For this, we introduce the following lemmas and relation: $\mathcal{R} \subset \mathrm{T} \times \mathrm{T} \times \Delta$, defined by the following inference rules. For convenience, we write the proposition $(x, y, z) \in \mathcal{R}$ as $x \mid y \rightsquigarrow z$.

$$t \mid t \rightsquigarrow \varepsilon$$

$$t \mid t' \rightsquigarrow \mu(t, t')$$

$$\frac{x \mid x' \rightsquigarrow \delta_x \qquad y \mid y' \rightsquigarrow \delta_y}{\tau_i(x, y) \mid \tau_j(x', y') \rightsquigarrow t_{\mu i \to j}(\delta_x, \delta_y)}$$

$$\frac{x \mid x' \rightsquigarrow \delta_x \qquad y \mid y' \rightsquigarrow \delta_y}{\tau_i(x, y) \mid \tau_i(x', y') \rightsquigarrow t_{\varepsilon i}(\delta_x, \delta_y)}$$

$$\frac{t \mid y' \rightsquigarrow \delta_y}{t \mid \tau_j(x', y') \rightsquigarrow t_{\pi_{\dashv j}}(x', \delta_y)}$$

$$\frac{t \mid x' \rightsquigarrow \delta_x}{t \mid \tau_j(x', y') \rightsquigarrow t_{\pi_{\vdash j}}(\delta_x, y')}$$

$$\frac{y \mid t \rightsquigarrow \delta_y}{\tau_i(x, y) \mid t \rightsquigarrow t_{\beta_{\dashv}}(\delta_y)}$$

$$\frac{x \mid t \rightsquigarrow \delta_x}{\tau_i(x, y) \mid t \rightsquigarrow t_{\beta_{\vdash}}(\delta_x)}$$

Figure 2: Inference rules for $\mathcal{R}$

The relation $\mathcal{R}$ is the relation between the input and the output of d, allowing for multiple images for a single input and thus getting rid of the $\min_w$ function in the diff process. We then use it as a proof device for simpler induction on diffs.

Lemma 3.3.1: $\forall t, t' : \mathrm{T}, \delta : \Delta, d(t, t') = \delta \Longrightarrow (t, t', \delta) \in \mathcal{R}$

Proof: By case disjunction on (t, t'). For every case, we suppose that $\delta = d(t, t')$ and we prove that $(t, t', \delta) \in \mathcal{R}$.

We then replace $d(t, t')$ by its expression and simplify the conditions for every case. From this, we can eliminate the two trivial cases involving constants on both sides, $(\kappa(x), \kappa(x))$ and $(\kappa(x), \kappa(y))$.
For all other cases, we apply another case disjunction on the output of $\min_w$. $\mathcal{R}$ is now trivially defined for every case of this new disjunction. Q.E.D.

Lemma 3.3.2: $\forall t, t' : \mathrm{T}, \delta : \Delta, (t, t', \delta) \in \mathcal{R} \Longrightarrow p(t, \delta) = t'$

Proof: By case disjunction on the different elements of $\mathcal{R}$. From then, one can trivially see from the definition of p that $(t, t', \delta) \in \mathcal{R} \Longrightarrow p(t, \delta) = t'$. Q.E.D.

We now prove the correctness of the pipeline:

Theorem 3.3.3 (Correctness): $\forall t, t' : \mathrm{T}, p(t, d(t, t')) = t'$

Proof: From Lemma 3.3.1 and Lemma 3.3.2, we see that $\forall t, t' : \mathrm{T}, \delta : \Delta, d(t, t') = \delta \Longrightarrow p(t, \delta) = t'$, thus $p(t, d(t, t')) = \delta$. Q.E.D.

4. Merging

4.1. Principle

If we take up the same arithmetical system as described in the diff/patch part, we can define the *merged diff* of δ_1 and δ_2, $\delta_3 = m(\delta_1, \delta_2)$, as the diff which, when added to the base tree t of both δ_1 and δ_2, includes both the changes described in δ_1 and those described in δ_2.

4.2. Algorithm

We thus define the merge function $m : \Delta \longrightarrow \Delta \longrightarrow \Delta$:

$$
\begin{aligned}
m(\varepsilon, x) &= x \\
m(x, \varepsilon) &= x \\
m(t_{\varepsilon i}(l, r), t_{\varepsilon i}(l', r')) &= t_{\varepsilon i}(m(l, l'), m(r, r')) \\
m\big(t_{\mu i \to j}(l, r), t_{\mu i \to j}(l', r')\big) &= t_{\mu i \to j}(m(l, l'), m(r, r')) \\
m\big(t_{\varepsilon i}(l, r), t_{\mu i \to j}(l', r')\big) &= t_{\mu i \to j}(m(l, l'), m(r, r')) \\
m\big(t_{\mu i \to j}(l', r'), t_{\varepsilon i}(l, r)\big) &= t_{\mu i \to j}(m(l, l'), m(r, r')) \\
m\big(t_{\varepsilon i}(l, r), \pi_{\dashv j}(t, \delta)\big) &= \pi_{\dashv j}(t, m(t_{\varepsilon i}(l, r), \delta)) \\
m\big(t_{\varepsilon i}(l, r), \pi_{\vdash j}(\delta, t)\big) &= \pi_{\vdash j}(m(t_{\varepsilon i}(l, r), \delta), t) \\
m\big(\pi_{\dashv j}(t, \delta), t_{\varepsilon i}(l, r)\big) &= \pi_{\dashv j}(t, m(t_{\varepsilon i}(l, r), \delta)) \\
m\big(\pi_{\vdash j}(\delta, t), t_{\varepsilon i}(l, r)\big) &= \pi_{\vdash j}(m(t_{\varepsilon i}(l, r), \delta), t) \\
m(t_{\varepsilon i}(\_, r), \beta_{\dashv}(\delta)) &= \beta_{\dashv}(m(r, \delta)) \\
m(t_{\varepsilon i}(l, \_), \beta_{\vdash}(\delta)) &= \beta_{\vdash}(m(l, \delta)) \\
m(\beta_{\dashv}(\delta), t_{\varepsilon i}(\_, r)) &= \beta_{\dashv}(m(r, \delta)) \\
m(\beta_{\vdash}(\delta), t_{\varepsilon i}(l, \_)) &= \beta_{\vdash}(m(l, \delta))
\end{aligned}
$$

$$m\left(\pi_{\dashv/\vdash i}(t,\delta), \pi_{\dashv/\vdash i}(t,\delta')\right) = \pi_{\dashv/\vdash i}(t, m(\delta,\delta'))$$

$$m\left(\beta_{\dashv/\vdash}(\delta), \beta_{\dashv/\vdash}(\delta')\right) = \beta_{\dashv/\vdash}(m(\delta,\delta'))$$

$$m(x,x) = x$$

One can see that the definition of m does not match the entirety of Δ^2. In cases not defined in the algorithm, a *merge conflict* has occured and an implementation of the algorithm should throw an error, indicating the location of the conflict to allow for fixing.

4.3. Correctness

5. Optimisation

5.1. ε-reduction

The first theoretical optimisation strategy is ε-reduction, that is folding the diffs that are equivalent to an absence of change into ε. Such an optimisation can easily be defined by the following $\varepsilon_R : \Delta \longrightarrow \Delta$ function:

$$\varepsilon_R(t_{\varepsilon i}(x,y)) = \begin{cases} \varepsilon \text{ if } \varepsilon_R(x) = \varepsilon_R(y) = \varepsilon \\ t_{\varepsilon i}(\varepsilon_R(x), \varepsilon_R(y)) \text{ else} \end{cases}$$

$$\varepsilon_R(\mu(x,y)) = \begin{cases} \varepsilon \text{ if } \varepsilon_R(x) = \varepsilon_R(y) \\ \mu(\varepsilon_R(x), \varepsilon_R(y)) \text{ else} \end{cases}$$

$$\varepsilon_R\left(t_{\mu i \to j}(x,y)\right) = t_{\mu i \to j}(\varepsilon_R(x), \varepsilon_R(y))$$

$$\varepsilon_R\left(\pi_{\dashv/\vdash i}(t,\delta)\right) = \pi_{\dashv/\vdash i}(t, \varepsilon_R(\delta))$$

$$\varepsilon_R\left(\beta_{\dashv/\vdash i}(\delta)\right) = \beta_{\dashv/\vdash i}(\varepsilon_R(\delta))$$

$$\varepsilon_R(\varepsilon) = \varepsilon$$

5.2. Diff optimality

6. Pratical considerations

6.1. Implementation strategies

The first strategy we used was treating the diffing problem as a shortest-path finding problem in a directed acyclic graph (DAG):

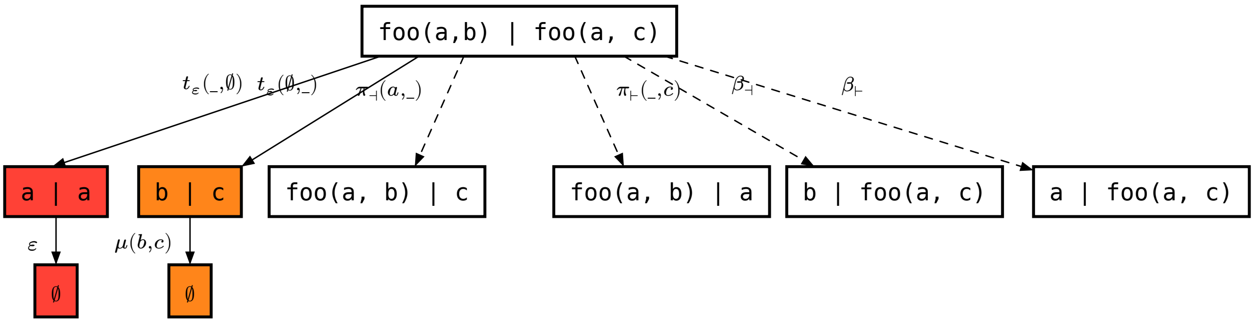

Figure 3: Graph formulation of the diff problem

Figure 3 displays the unfolding of the thus conceived diffing process, implemented using a modified version of the A* algorithm. All nodes shown in the figure were pushed onto the min-heap at some point and dotted edges indicate unvisited paths. Colours indicate that such nodes were constructed by recursive calls to the `diff` function. The heap we use minimises $f(n) = g(n) + h(n)$ where g and h are calculated as follows: $g(n)$ is 0 for ε and t_ε (and necessarily for the initial node), $|x| + |y|$ for $\mu(x, y)$ and 1 for other constructors. If corresponds to the previously defined w function when applied to the entire graph. The heuristic function h is defined by $h(l, r) = \min(|l|, |r|)$, where l and r are respectively the left and right trees the diff is processing. We also have $h(\emptyset) = 0$ and when dealing with recursive (i.e. binary) constructors, the smallest heuristic value is kept.

When compared with a rather naive implementation (with memoisation of already-diffed nodes as sole optimisation), this method has shown to greatly reduce (approximately tenfold) the time needed to diff the same file pairs.

6.2. Formatting preservation

7. Performance

7.1. Methodology

7.2. Results

8. Related work

9. Further research

10. Conclusion

A Some proofs

A.1 Lemma 2.1