Best Title in the Universe

42

Victor Cacciari Miraldo   Wouter Swierstra

University of Utrecht

{v.cacciarimiraldo,w.s.swierstra} at uu.nl

Abstract

stuff

Categories and Subject Descriptors   D.1.1 [*look*]: for—this

General Terms   Haskell

Keywords   Haskell

1.   Introduction

The majority of version control systems handle patches in a non-structured way. They see a file as a list of lines that can be inserted, deleted or modified, with no regard to the semantics of that specific file. The immediate consequence of such design decision is that we, humans, have to solve a large number of conflicts that arise from, in fact, non conflicting edits. Implementing a tool that knows the semantics of any file we happen to need, however, is no simple task, specially given the plethora of file formats we see nowadays.

This can be seen from a simple example. Lets imagine Alice and Bob are iterating over a cake's recipe. They decide to use a version control system and an online repository to keep track of their modifications.
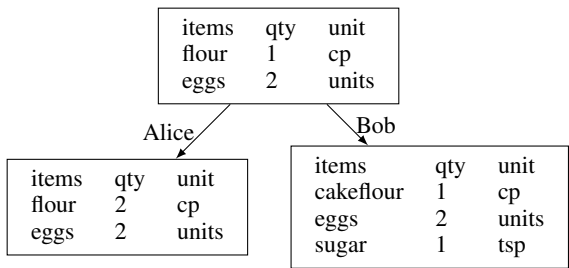

Figure 1. Sample CSV files

Lets say that both Bob and Alice are happy with their independent changes and want to make a final recipe. The standard way to track differences between files is the `diff3` [FIXBIB] unis tool. Running `diff3 Alice.csv O.csv Bob.csv` would result in the output presented in figure 2. Every tag ==== marks a difference. Three

locations follows, formatted as `file:line type`. The change type can be a *Change*, *Append* or *Delete*. The first one, says that file 1 (`Alice.csv`) has a change in line 2 (`1:2c`) which is `flour, 2 , cp`; and files 2 and 3 have different changes in the same line. The tag `====3` indicates that there is a difference in file 3 only. Files 1 and 2 should append what changed in file 3 (line 4).

```
====
1:2c
  flour, 2  , cp
2:2c
  flour, 1  , cp
3:2c
  cakeflour, 1  , cp
====3
1:3a
2:3a
3:4c
  sugar, 1  , tsp
```

Figure 2. Output from `diff3`

If we try to merge the changes, `diff3` will flag a conflict and therefore require human interaction to solve it, as we can see by the presence of the ==== indicator in its output. However, Alice's and Bob's edits, in figure 1 do *not* conflict, if we take into account the semantics of CSV files. Although there is an overlapping edit at line 1, the fundamental editing unit is the cell, not the line.

We propose a structural diff that is not only generic but also able to track changes in a way that the user has the freedom to decide which is the fundamental editing unit. Our work was inspired by [**?**] and [**?**]. We did extensive changes in order to handle structural merging of patches. We also propose extensions to this algorithm capable of detecting purely structural operations such as refactorings and cloning.

The paper begins by exploring the problem, generically, in the Agda [FIXBIB] language. Once we have a provably correct algorithm, the details of a Haskell implementation of generic diff'ing are sketched. To open ground for future work, we present a few extensions to our initial algorithm that could be able to detect semantical operations such as *cloning* and *swapping*.

Contributions

- *Study of a more algebraic patch theory.*
- *Agda model.*
- *Haskell Prototype.*

Background

2. Structural Diffing

Alice and Bob were both editing a CSV file which represents data that is isomorphic to $[[Atom\ String]]$, where $Atom\ a$ is a simple tag that indicates that as should be treated abstractly, that is, either they are equal or different, we will not open these values to check for structural changes.

As we are tracking differences, there are a few operations that are inherent to our domain, such as: inserting; deleting; copying and updating. When we say *structural diffing*, however, we add another option to this list. Now we will also be able to go down the structure of some object and inspect its parts. To illustrate this, let us take Alice's change as in figure 1, her changes to the file could be described, structurally, as:

I) Copy the first line;

II) Enter the second line;
   i) Copy the first field;
   ii) Enter the second field;
     • Update atom `"1"` for atom `"2"`.
   iii) Copy the third field;

III) Copy the third line.

IV) Finish.

In figure 3 we show the patch that corresponds to that.
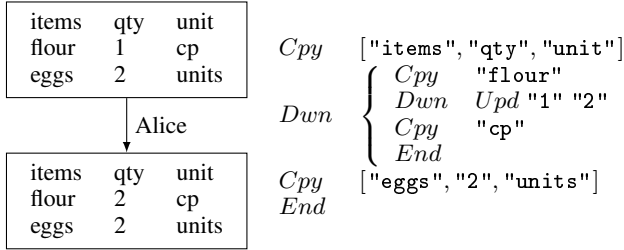
items	qty	unit
flour	1	cp
eggs	2	units

↓ Alice

items	qty	unit
flour	2	cp
eggs	2	units

Cpy $\quad$ $[$"items", "qty", "unit"$]$

Dwn $\begin{cases} Cpy & \text{"flour"} \\ Dwn & Upd\ \text{"1" "2"} \\ Cpy & \text{"cp"} \\ End \end{cases}$

Cpy $\quad$ $[$"eggs", "2", "units"$]$
End

Figure 3. Alice's Patch

Consider now Bob's structural changes to the CSV file[1]. If you overlap both, you should notice that there is Upd operation on top of another. This was in fact expected given that Alice and Bob performed changes in disjoint parts of the CSV file.

---

[1] Exercise to the reader! Clue: the last two operations are $Ins\ [$"sugar", "1", "tsp"$]\ End$

2.1 Context Free Datatypes

Although our running example, of CSV files, has type $[[Atom\ String]]$, lists of a themselfes are in fact the least fixed point of the functor $X \mapsto 1 + a \times X$. Which is a *context-free type*, in the sense of [**?**]. For it is constructed following the grammar CF of context free types with a deBruijn representation for variables.

$$CF ::= 1 \mid 0 \mid CF \times CF \mid CF + CF \mid \mu\ CF \mid \mathbb{N}$$

In Agda, the CF universe is defined by:

```
data U : ℕ → Set where
    u0  : {n : ℕ} → U n
    u1  : {n : ℕ} → U n
    _⊕_ : {n : ℕ} → U n → U n → U n
    _⊗_ : {n : ℕ} → U n → U n → U n
    β   : {n : ℕ} → U (suc n) → U n → U n
    μ   : {n : ℕ} → U (suc n) → U n
    vl  : {n : ℕ} → U (suc n)
    wk  : {n : ℕ} → U n → U (suc n)
```

We stress that one of the main objectives of this project is to release a solid diffing and merging tool that has formal guarantees, written in Haskell. The universe of user-defined Haskell types is smaller than context free types; in fact, we have fixed-points of sums-of-products. Therefore, we should be able to apply the knowledge acquired in Agda directly in Haskell. In fact, we did so! With a few adaptations here and there, to make the type-checker happy, the Haskell code is almost a direct translation, and will be discussed in section 6.

Stating the language of our types is not enough. We need to specify its elements too. Fixed-points make things a bit more complicated, therefore we refer the reader to [**?**] for a more in-depth explanation on the topic. In Agda, the elements of U are defined by:

```
data ElU : {n : ℕ} → U n → Tel n → Set where
    void : {n : ℕ}{t : Tel n} → ElU u1 t
    inl  : {n : ℕ}{t : Tel n}{a b : U n}(x : ElU a t) → ElU (a ⊕ b) t
    inr  : {n : ℕ}{t : Tel n}{a b : U n}(x : ElU b t) → ElU (a ⊕ b) t
    _,_  : {n : ℕ}{t : Tel n}{a b : U n} → ElU a t → ElU b t → ElU (a ⊗ b) t
    top  : {n : ℕ}{t : Tel n}{a : U n}  → ElU a t → ElU vl (tcons a t)
    pop  : {n : ℕ}{t : Tel n}{a b : U n} → ElU b t → ElU (wk b) (tcons a t)
    mu   : {n : ℕ}{t : Tel n}{a : U (suc n)} → ElU a (tcons (μ a) t) → ElU (μ a) t
    red  : {n : ℕ}{t : Tel n}{F : U (suc n)}{x : U n}
           → ElU F (tcons x t)
           → ElU (β F x) t
```

The Tel index is the telescope in which to look for the instantiation of type-variables. A value $(v\ :\ ElU\ \{n\}\ ty\ t)$ reads roughly as: a value of type ty with n variables, applied to n types t with at most $n-1$ variables. We need this decrease of type variables to convince the termination checker that our code is ok.

Our universe choice supports a great deal of generic definitions; we have generic decidable equality, generic mapping, generic childrens, etc. Explaining

2.2 Patches over a Context Free Type

A patch over A is an object that describe possible changes that can be made to objects of type A. The high-level idea is that diffing two

objects $a_1, a_2 : A$ will produce a patch over A, whereas applying a patch over A to an object will produce a $Maybe\ A$. It is interesting to note that application can not be made total. Let's consider $A = X + Y$, and now consider a patch $(Left\ x) \xrightarrow{p} (Left\ x')$. What should be the result of applying p to a $(Right\ y)$? It is undefined!

In order to simplify the presentation, we are gonna explicitly name variables and write our types in a more mathematical fashion, other than the Agda encoding. As we discussed earlier, a patch is an object that track differences in a given type. Different types will allow for different types of changes.

Definition 2.1 (Simple Patch)**.** *We define a (simple) patch $D\ ty$ by induction on ty as:*

$$
\begin{aligned}
D\ 0 &= 0 \\
D\ 1 &= 1 \\
D\ (x \times y) &= D\ x \times D\ y \\
D\ (x + y) &= (D\ x + D\ y) + 2 \times (x \times y) \\
D\ (\mu X.F\ X) &= \mu X.(1 \\
&+\quad D\ (F\ 1) \times X \\
&+\quad 2 \times (F\ 1) \times X \\
&)
\end{aligned}
$$

Where 1 and 0 are the usual terminal and initial objects of a given category.

Let's see the coproduct case in more detail. There are four different possibilities for the changes seen in a coproduct, just like there are four different combinations of constructors for two objects of type $Either\ a\ b$. The first and second options, namely $D\ x$ and $D\ y$ track differences of a $Left\ a$ into a $Left\ a'$ and a $Right\ b$ into a $Right\ b'$, respectively. The other possibilities are representing a $Left\ a$ becoming a $Right\ b$ or vice-versa. The other branches are straight-forward.

Producing Patches Definition 2.1 provides us with some intuition on how one would define patches for a given datatype. The actual definition (figure 4) is more complicated, though. The Diff type takes one parameter, used to give a free-monad [FIXBIB] structure, and two indexes which indicate the type for which that diff is intented. We then define, with $\perp_p = \lambda\_\_ \to \perp$:

```
Patch : {n : ℕ} → Tel n → U n → Set
Patch t ty = D ⊥_p t ty
```

Our first goal is, of course, to produce patches, or to differentiate between to objects of the same type. We can do that generically through

```
gdiff : {n : ℕ}{t : Tel n}{ty : U n}
     → ElU ty t → ElU ty t → Patch t ty
gdiff {ty = vl} (top a) (top b)      = D-top (gdiff a b)

gdiff {ty = wk u} (pop a) (pop b) = D-pop (gdiff a b)

gdiff {ty = β F x} (red a) (red b) = D-β (gdiff a b)

gdiff {ty = u1} void void = D-void

gdiff {ty = ty ⊗ tv} (ay , av) (by , bv)
   = D-pair (gdiff ay by) (gdiff av bv)

gdiff {ty = ty ⊕ tv} (inl ay) (inl by) = D-inl (gdiff ay by)
gdiff {ty = ty ⊕ tv} (inr av) (inr bv) = D-inr (gdiff av bv)
gdiff {ty = ty ⊕ tv} (inl ay) (inr bv) = D-setl ay bv
gdiff {ty = ty ⊕ tv} (inr av) (inl by) = D-setr av by

gdiff {ty = μ ty} a b = D-mu (gdiffL (a :: []) (b :: []))
```

Here the `gdiffL` function is virtually the same as presented in [**?**], with the extra case for traversing down a constructor.

- *wrap it up?*

Definition 2.2 (Defined)**.** *We say that a patch p_a is defined for an input a iff there exists an object a' such that:*

$$apply\ p_a\ a \equiv Just\ a'$$

Fixed Points The treatment for fixed points has to be made uniform, somehow, if we want a generic algorithm by the end of the day. What makes fixed points different than regular algebraic types is that they can grow or shrink arbitralily, and our diff function has to take that into account.

Recalling the fixed point clause of simple patches (def 2.1),

$$
\begin{aligned}
D\ (\mu X.F\ X) &= \mu X.(1 \\
&+\quad D\ (F\ 1) \times X \\
&+\quad 2 \times (F\ 1) \times X \\
&)
\end{aligned}
$$

it is straight forward to see that the $D\ (\mu X.F\ X)$ is isomorphic to a list with three recursive constructors and a non-recursive one. Following the edit operations studied by Löh[**?**], we have an *insert*, a *delete* and a *end* edit operations. The big difference is that instead of copying, we have a constructor that track changes inside a constructor of $\mu X.F\ X$, we call this a *down* edit operation.

We heavily rely on the facth that $\mu X.F\ X \approx F\ 1 \times [\mu X.F\ X]$, that is, any inhabitant of a fixed-point type can be seen as a non-recursive head and a list of recursive children, or, expressed in our generic setting:

```
Openμ : {n : ℕ} → Tel n → U (suc n) → Set
Openμ t ty = ElU ty (tcons u1 t) × List (ElU (μ ty) t)

μ-open : {n : ℕ}{t : Tel n}{ty : U (suc n)}
     → ElU (μ ty) t → Openμ t ty

μ-close : {n : ℕ}{t : Tel n}{ty : U (suc n)}
     → Openμ t ty → Maybe (ElU (μ ty) t × List (ElU (μ ty) t))
```

Although we could have used vectors of a fixed length and made this a total isomorphism, we would have more problems than benefits. This will be discussed in section 5.2. Nonetheless, an important soundness result has been proven:

```
mutual
  data D {a}(A : {n : ℕ} → Tel n → U n → Set a) : {n : ℕ} → Tel n → U n → Set a where
    D-A    : {n : ℕ}{t : Tel n}{ty : U n} → A t ty → D A t ty

    D-void : {n : ℕ}{t : Tel n} → D A t u1
    D-inl  : {n : ℕ}{t : Tel n}{a b : U n}
           → D A t a → D A t (a ⊕ b)
    D-inr  : {n : ℕ}{t : Tel n}{a b : U n}
           → D A t b → D A t (a ⊕ b)
    D-setl : {n : ℕ}{t : Tel n}{a b : U n}
           → ElU a t → ElU b t → D A t (a ⊕ b)
    D-setr : {n : ℕ}{t : Tel n}{a b : U n}
           → ElU b t → ElU a t → D A t (a ⊕ b)
    D-pair : {n : ℕ}{t : Tel n}{a b : U n}
           → D A t a → D A t b → D A t (a ⊗ b)
    D-mu : {n : ℕ}{t : Tel n}{a : U (suc n)}
         → List (Dμ A t a) → D A t (μ a)
    D-β : {n : ℕ}{t : Tel n}{F : U (suc n)}{x : U n}
         → D A (tcons x t) F → D A t (β F x)
    D-top : {n : ℕ}{t : Tel n}{a : U n}
         → D A t a → D A (tcons a t) vl
    D-pop : {n : ℕ}{t : Tel n}{a b : U n}
         → D A t b → D A (tcons a t) (wk b)

  data Dμ {a}(A : {n : ℕ} → Tel n → U n → Set a) : {n : ℕ} → Tel n → U (suc n) → Set a where
    Dμ-A   : {n : ℕ}{t : Tel n}{a : U (suc n)} → A t (μ a) → Dμ A t a
    Dμ-ins : {n : ℕ}{t : Tel n}{a : U (suc n)} → ValU a t → Dμ A t a
    Dμ-del : {n : ℕ}{t : Tel n}{a : U (suc n)} → ValU a t → Dμ A t a
    Dμ-cpy : {n : ℕ}{t : Tel n}{a : U (suc n)} → ValU a t → Dμ A t a
    Dμ-dwn : {n : ℕ}{t : Tel n}{a : U (suc n)} → ValU a t → D A t (β a u1) → Dμ A t a
```

Figure 4. Complete Definition of D

```
μ-close-resp-arity
  : {n : ℕ}{t : Tel n}{ty : U (suc n)}{a : ElU (μ ty) t}
    {hdA : ElU ty (tcons u1 t)}{chA l : List (ElU (μ ty) t)}
  → μ-open a ≡ (hdA , chA)
  → μ-close (hdA , chA ++ l) ≡ just (a , l)
```

2.3 The Cost Function

- *the cost function should satisfy a few properties, such as: if x and y come from the same constructor, then $cost(diff\,xy) \leq cost(Delx :: Addy :: End)$. Otherwise, $gdiffL$ will always choose $DmuDwn$ first.*

3. A Category of Patches

To Research!

- *Define patch composition, prove it makes a category.*
- *But then... does it make sense to compute the composition of patches?*
- *In a vcs setting, we always have the intermediate files that originated the patches, meaning that composition can be defined semantically by: $apply(p \cdot q) \equiv applyq \circ applyp$, where $\circ$ is the Kleisli composition of $+1$.*
- *This gives me an immediate category... how usefull is it?*

4. Patch Propagation

Let's say Bob and Alice perform edits in a given object, which are captured by patches p and q. In the version control setting, the natural question to ask is *how do we join these changes*.

There are two solutions that could possibly arise from this question. Either we group the changes made by p and by q (as long as they are compatible) and create a new patch to be applied on the source object, or, we calculate how to propagate the changes of p over q and vice-versa. Figure 5 illustrates these two options.
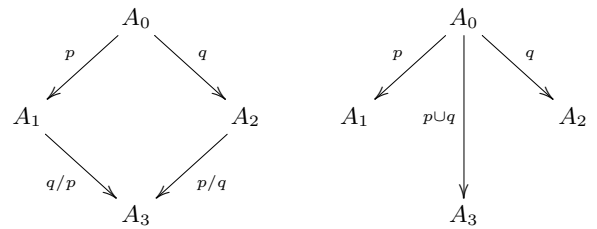

Figure 5. Residual Square on the left; three-way-merging on the right

The residual p/q of two patches p and q only makes sense if both p and q are aligned, that is, are defined for the same input. It captures the notion of incorporating the changes made by p in an object that has already been modified by q.

We chose to use the residual notion, as it seems to have more structure into it. Not to mention we could define $p \cup q \equiv (q\,p) \cdot p \equiv (p/q) \cdot q$. Unfortunately, however, there exists conflicts we need to take care of, which makes everything more complicated.

In an ideal world, we would expect the residual function to have type $D\ a \to D\ a \to Maybe\ (D\ a)$, where the partiality comes from receiving two non-aligned patches.

But what if Bob and Alice changes the same cell in their CSV file? Then it is obvious that someone (human) have to chose which value to use in the final, merged, version.

For this illustration, we will consider the conflicts that can arise from propagating the changes Alice made over the changes already made by Bob, that is, p_{alice}/p_{bob}.

- If Alice changes a_1 to a_2 and Bob changed a_1 to a_3, with $a_2 \neq a_3$, we have an *update-update* conflict;
- If Alice adds information to a fixed-point, this is a *grow-left* conflict;
- When Bob added information to a fixed-point, which Alice didn't, a *grow-right* conflict arises;
- If both Alice and Bob add different information to a fixed-point, a *grow-left-right* conflict arises;
- If Alice deletes information that was changed by Bob we have an *delete-update* conflict;
- Last but not least, if Alice changes information that was deleted by Bob we have an *update-delete* conflict.

Above we see two distinct conflict types. An *update-update* conflict has to happen on a coproduct type, whereas the rest are restricted to fixed-point types. In Agda,

```
data C : {n : ℕ} → Tel n → U n → Set where
  UpdUpd : {n : ℕ}{t : Tel n}{a b : U n}
             → ElU (a ⊕ b) t → ElU (a ⊕ b) t → ElU (a ⊕ b) t
             → C t (a ⊕ b)
  DelUpd : {n : ℕ}{t : Tel n}{a : U (suc n)}
             → ValU a t → ValU a t → C t (μ a)
  UpdDel : {n : ℕ}{t : Tel n}{a : U (suc n)}
             → ValU a t → ValU a t → C t (μ a)
  GrowL  : {n : ℕ}{t : Tel n}{a : U (suc n)}
             → ValU a t → C t (μ a)
  GrowLR : {n : ℕ}{t : Tel n}{a : U (suc n)}
             → ValU a t → ValU a t → C t (μ a)
  GrowR  : {n : ℕ}{t : Tel n}{a : U (suc n)}
             → ValU a t → C t (μ a)
```

- *Pijul has this notion of handling a merge as a pushout, but it uses the free co-completion of a rather simple category. This doesn't give enough information for structured conflict solving.*
- *BACK THIS UP!*

4.1 Incorporating Conflicts

In order to track down these conflicts we need a more expressive patch data structure. We exploit D's parameter for that matter. This approach has the advantage of separating conflicting from conflict-free patches on the type level, guaranteeing that we can only *apply* conflict-free patches.

The type of our residual[2]. operation is:

```
_/_ : {n : ℕ}{t : Tel n}{ty : U n}
    → Patch t ty → Patch t ty → Maybe (D C t ty)
```

We reiterate that the partiality comes from the fact the residual is not defined for non-aligned patches. We chose to make a partial function instead of receiving a proof of alignment purely for pratical purposes. Defining alignment for our patches is very complicated.

---

[2] Our residual operation does not form a residual as in the Term Rewriting System sense[**?**]. It might, however, satisfy interesting properties. This is left as future work for now

The attentive reader might have noticed a symmetric structure on conflicts. This is not at all by chance. In fact, we can prove that the residual of p/q have the same (modulo symmetry) conflicts as q/p. This proof goes in two steps. Firstly, residual-symmetry proves that the symmetric of the conflicts of p/q appear in q/p, but this happens modulo a function. We then prove that this function does not introduce any new conflicts, it is purely structural.

```
residual-symmetry-thm
  : {n : ℕ}{t : Tel n}{ty : U n}{k : D C t ty}
  → (d1 d2 : Patch t ty)
  → d1 / d2 ≡ just k
  → Σ (D C t ty → D C t ty)
     (λ op → d2 / d1 ≡ just (D-map C-sym (op k)))

residual-sym-stable : {n : ℕ}{t : Tel n}{ty : U n}{k : D C t ty}
  → (d1 d2 : Patch t ty)
  → d1 / d2 ≡ just k
  → forget <M> (d2 / d1) ≡ just (map (↓-map-↓ C-sym) (forget k))
```

Here $<M>$ denotes the Kleisli composition of the $Maybe$ monad and $\downarrow-map-\downarrow$ takes care of the indexes.

Now, we can compute both p/q and q/p at the same time. It also backs the intuition that using residuals or patch commutation (as in darcs) is not significantly different.

This means that p/q and q/p, although different, have the same conflicts (up to symmetry).

4.2 Solving Conflicts

- *This is highly dependent on the structure.*
  - *some structures might allow permutations, refactorings, etc... whereas others might not.*
- *How do we go generic? Free-monads to the rescue!*

5. Summary and Remarks

5.1 Sharing of Recursive Subterms

- If we want to be able to share recursive subexpressions we need a mutually recursive approach.
- Or, this will be handled during conflict solving. See refactoring.

5.2 Remarks on Type Safety

- only the interface to the user can be type-safe, otherwise we don't have our free-monad multiplication.

6. A Haskell Prototype

- *throw hs-diff in github before the deadline!*

7. Sketching a Control Version System

- Different views over the same datatype will give different diffs.
- **newtype** annotations can provide a gread bunch of control over the algorithm.
- Directories are just rosetrees...

8. Related Work

To Research!

- *Check out the antidiagonal with more attention: http://blog.sigfpe.com/2007/09/type-of-distinct-pairs.html*

- **ANS:** *Diffing and Antidiagonals are fundamentally different. The antidiagonal for a type T is a type X such that there exists $X \to T^2$. That is, X produces two **distinct** T's, whereas a diff produces a T given another T!*

9. Conclusion

- This is what we take out of it.

References

[] T. Altenkirch, C. Mcbride, and P. Morris. Generic programming with dependent types. In *Spring School on Datatype Generic Programming*. Springer-Verlag, 2006.

[] M. Bezem, J. Klop, R. de Vrijer, and Terese. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci*, 337:217–239, 2005.

[] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, Oct. 1977.

[] E. Lempsink, S. Leather, and A. Löh. Type-safe diff for families of datatypes. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*, WGP '09, pages 61–72, New York, NY, USA, 2009. ACM.

[] A. Mestanogullari, S. Hahn, J. K. Arni, and A. Löh. Type-level web apis with servant: An exercise in domain-specific generic programming. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, pages 1–12, New York, NY, USA, 2015. ACM.

[] T. Rendel and K. Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing. *SIGPLAN Not.*, 45(11):1–12, 2010.

[] J. Shaw. boomerang. `http://hackage.haskell.org/package/boomerang`, 2014. Acessed: November 2015.

[] S. Tieleman. Formalisation of version control with an emphasis on tree-structured data. Master's thesis, Universiteit Utrecht, Aug. 2006.

[] M. Vassena. Svc, a prototype of a structure-aware version control system. Master's thesis, Universiteit Utrecht, 2015.