Version Control Systems: Diffing with Structure
Thesis

Giovanni Garufi
5685109

Supervisors:
Wouter Swierstra
Victor Cacciari Miraldo

Department of Computing Science
University of Utrecht

December 18, 2017

Contents

1 Introduction

Version control systems (VCS) have been steadily becoming an ubiquitous and central tool in programming. Many big projects, with hundreds of collaborators, fundamentally rely on these kind of tools to allow collaborators to interact with one another and to keep a structured log of the units of change. At the heart of most modern VCS is the Unix *diff* utility. This tool computes a line-by-line difference between two files of text, determining the smallest set of insertions or deletions of lines that transform one file into the other. This sequence of transformation produced by *diff* is called an edit script. When two collaborators have modified the same source file in independent ways and the VCS wants to reconcile the two independent changes into a single one it will attempt to produce an single edit script that encompasses both changes, this is commonly referred to as a merge. This operation is clearly not always possible: if two collaborators have modified the same line the two resulting edit scripts will "overlap", it is not clear which one of the two modifications should be picked. The algorithm used to calculate these merges in most VCS is *diff*3, which works by calculating the two edit-scripts with *diff*, and attempting to reconcile them into a single script.

In general, some conflicts will always require a manual intervention: when two people change the same thing in two different ways there is no general way of deciding which should be the resulting transformation; however the diff tool makes a big assumption in considering lines to be the basic units in which change is observable.

The shortcomings of the approach in *diff*3 can be seen in this simple example. Suppose we have the following function in any lisp-like language:

```
(defn head [l]
  (first l))
```

Suppose we make two independent modifications, the first of which adds a default parameter to be returned in case the list is empty, and the second one that changes the name of the function from `head` to `fst`.

```
(defn head [l, d]
  (if (nil? l)
    (d)
    (first l)))

(defn fst [l]
  (first l))
```

When attempting to reconcile these two changes with the *diff*3 algorithm we will run into problems; despite the two patches are modifying disjoint pieces of the actual code, *diff*3 will output a conflict. The reason is that both changes touch the same line; this is a nuisance as manual intervention is required to solve the conflicts. Furthermore it also introduces some level of non-determinism, as the presence of conflicts may depend on things like indentation instead of being a fundamental property of the transformation. The main idea of this research is to design an alternative to *diff* which offers a finer grained control over the units of change. By attempting to extend the *diff* algorithm to operate on an AST that represents the parsed program, we are able to focus on smaller units of change; this allows us to produce more accurate patches.
The drawback of this approach is that it moves the problem from the world of strings to the world of trees, and the problem of computing a patch between two elements suddenly becomes computationally more expensive. Approaches similar to this one have already been explored in previous literature [5], [4] and [6]. In these papers, the problem of computing the difference between two trees was always reduced to the problem of computing the difference of a flattened representation of the tree. While the flattened representation makes the problem of computing a difference easier, it makes reconstructing a valid tree from the representation more complex, and patches are more likely to generate ill-structured data. The novelty in the work of Miraldo, Dagand and Swierstra [7] lies in the idea of enforcing a structure-preserving, type-directed approach. On one hand structural information is directly encoded in the patches, so that applying a patch will always produce well-formed code. On the other hand, the type information encoded in the grammar of the AST is exploited in the creation of a patch, making the process more efficient.
The paper introduces a theoretical and practical framework to define and compute patches between structured data. This framework is generic and makes extensive use of dependent types in order to guarantee structure preserving transformations.

1.1 Overview

The chapters are structured as following: We start by showing a full Haskell implementation of the algorithm presented by Miraldo et al. [7], which is presented in Agda in the original paper. The algorithm is implemented generically and described through dependent types in the paper; one of the main contributions of this thesis is to provide a Haskell implementation that

is suitable to run experiments on real-world data. Haskell is scheduled to land full support for dependent types in the next couple of years [10], in the meanwhile, they can only be partially supported through some ad-hoc techniques. Both Generics and Dependent Types require some language extensions and machinery which are non trivial in Haskell.
The following section starts with an introduction to dependent types in Haskell as they will be crucial in encoding the structure we want to express in our data types and their transformations. Essentially we want to characterize patches by the transformation they operate on the source code (e.g. the patch that adds an extra argument to a function) as such, we need dependent types to reflect onto the type system the action of a patch on a certain value. This will assure that any code produced by the algorithm is structurally valid by construction.
Following dependent types we will need to introduce sums of products: these give us a general way to view types and will allow us to define an algorithm that is independent of the representation of the AST for the language we are treating. In this regard there is a fine balance between having a core algorithm which is generic and can be applied to any language, and the use of domain-specific strategies to guide the generation of patches by using knowledge specific to the language in question. One of the goals of this thesis is to investigate this balance: how necessary are domain-specific diffing strategies in order to keep the combinatorial explosion in check? Despite the generality of the algorithm, we have instantiated it for the Clojure programming language [9]; this choice is motivated by the general simplicity of parsing languages that derive from LISP and by the need to select a language that is popular enough to have large active projects, available on Github, that will provide us a good sample of data to test.
The specification given by Miraldo et al. [7] is completely non-deterministic. After presenting the algorithm for a type-directed diff, alongside its Haskell implementation, we will analyze the performance of the non-deterministic specification and conclude that it is still to slow for real-world data. To solve this problem we will introduce different heuristics to guide the process of patch generation and analyze their performance and shortcomings. Finally we will introduce the notion of *disjointedness*, a predicate that attempts to capture the intuition that two patches that "touch different things" should be mergeable. We will use this predicate to run experiments on conflicts gathered from public repositories on Github and compare the amount of merge conflicts obtained by our approach compared to *diff*3

2 Background

With time, Haskell's type system has kept evolving from its humble Hindley-Miller origins and through the use of different language extensions it has gained the ability to express more complex types. In particular, many efforts have gone to add partial support for dependently typed programming in the latest years.
One major stepping stone in this direction is the DataKinds ([3]) extension which duplicates an ordinary data type, such as

```
data Nat = Z | S Nat
```

at the kind level, this means that from this declaration we automatically get two new types, namely `Z` of kind `Nat` and `S` of kind `Nat\ -\textgreater{}\ Nat`. We can use the `Nat` kind to index generalized algebraic data types (GADTs) in a way that allows us to create an analogue of a dependent type. In the case of `Nat`, we can use it to define a GADT for vectors of a given length.

```
data Vec :: * -> Nat -> * where
  Vn :: Vec x Z
  Vc :: x -> Vec x n -> Vec x (S n)
```

Such a vector is either the empty vector, which is indexed by `Z`, or a vector which is built by adding an element of type `x` in front of a vector of `xs` of length `n`, yielding a vector of `xs` of length `S\ n`.

This allows us to define principled analogues of some functions which operate on lists. The infamous `head` function will crash our program when passed an empty list; equipped with these `Vec`, we can rule this out by construction.

This is how we can define a `head` function on vectors.

```
head :: Vec x (S n) -> x
head (Vc h t) = h
```

Informally, we are saying that the `head` function takes as argument a vector with length strictly greater than 0. In this way, if we try to pass an empty vector to `head` we will get a compile time error instead of the usual run time one.

Another extension which plays a crucial role in dependent types is `TypeFamilies`: informally it allows us to write functions which operate on types, we will use this to define concatenation between `Vec`s.

The following type family can be seen as a function that takes two types of kind `Nat` and returns another type of kind `Nat` representing the result of adding those two types.

```
type family (m :: Nat) :+ (n :: Nat) :: Nat where
  Z     :+ n = n
  (S m) :+ n = S (m :+ n)
```

Equipped with this type family we can now define concatenation between vectors.

```
vappend :: Vec x n -> Vec x m -> Vec x (n :+ m)
vappend Vn          ys = ys
vappend (x `Vc` xs) ys = x `Vc` (vappend xs ys)
```

One interesting thing to note is that up to this point, we never use the `Nat` part of a vector at run time. That information is only used at compile time to check that everything "lines up" the way it should be, but it could actually be erased at runtime.

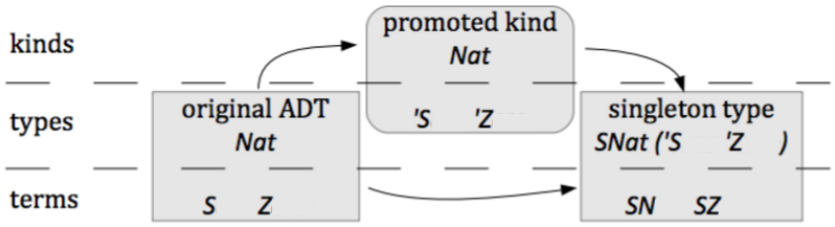
Suppose we want to write a `split` function, this function takes an `n` of kind `Nat`, a vector of length `n :+ m` and splits it into a pair of vectors, with respectively `n` and `m` elements.

The first problem we incur in is that we can not pass something of kind `Nat` to our `split` function, in fact the types `Z` and `S` have no inhabitants, so we can not construct any term of those types. Furthermore, we want to express that this `n` of kind `Nat` that we pass as a first argument, is the same `n` as in the vector length. The idea is to wrap this type into a singleton data type, giving us a dynamic container of the static information.

```
data SNat :: Nat -> * where
  SZ ::           SNat Z
  SN :: SNat n -> SNat (S n)
```

The name singleton comes from the fact that each type level value of kind `Nat` (namely `Z` or `S` applied to another type of kind `Nat`) has a single representative in the `SNat` type.

To sum things up: DataKinds extension promotes members of the type `Nat` to inhabitants of the kind `Nat`. Singletons allow us to take one step back in this ladder, associating to every thing of kind `Nat` a term from the singleton type `SNat`. The following picture, borrowed from Eisenberg et al. [2], gives a good representation of this process.



We can think of the DataKinds extension as a way of embedding dynamic information into the static fragment of the language. Singletons, on the other hand, are a way to reflect this static information back to the dynamic level, and make run-time decisions based on the types we obtain.

Singletons solve the two problems outlined above: they have kind $*$ and contain a `Nat` that we can later refer to in our function definition. We can now define `split` as follows:

```
split :: SNat n -> Vec x (n :+ m) -> (Vec x n, Vec x m)
split SZ     xs          = (Vn, xs)
split (Sn n) (x `Vc` xs) = (x `Vc` ys, zs)
  where
    (ys, zs) = split n xs
```

With these three tricks up our sleeve (data kind promotion, type level functions and singletons) we can emulate some of the features that are present in dependently typed languages such as Agda. These features allow us to emulate explicit dependent quantification. We can actually go even further in Haskell: Lindley et McBride [1] show us how to emulate implicit types via type classes and ultimately how all kinds of quantification, modulo some boilerplate, are possible in Haskell.

2.1 Sum of Products

The basic idea of the Sum of Products (SOP) approach is to define a "normal form" for all generic representations of data types and to define functions that act on this form. It is, in some sense, similar to the Disjunctive Normal Form for propositional logic in the sense that every proposition can be expressed in DNF and we can write theorems (functions) which assume the expression is represented in DNF. The bulk idea is to view each type as a choice between a constructor and all the arguments that are passed to that constructor. It is useful to think about the two different levels: on the first

level we make a choice about which constructor to pick; this corresponds to a sum over all the constructors of the type. On the second level, we are choosing a certain number of arguments to feed to that constructor and this can be viewed as a list, or product, of those arguments. Clearly the products will depend on the choice of constructor, each of which could take a different number of arguments of possibly different types. This motivates the need to represent the product as some sort of heterogeneous list. A constructor can also take no arguments, in which case we can simply use an empty list to represent that, but can also take an argument of the same type it is trying to construct (like the `S` constructor from the previous section). In this case, the recursive argument can itself be encoded as an SOP and the same encoding can be used all the way down to the leaves.

Since every data type can be encoded as an SOP, we can write functions that act on this representation and regard them as generics. For every data type we can first convert it to the SOP representation and then pass it to the desired function. Since the SOP representation is isomorphic to the original type, we can eventually reconstruct the desired term after we are done working with the representation. This encoding is presented by Loh et al. [8] and represents one of the numerous different ways to approach generic programming in Haskell. In the remainder of this section we will try to formalize the intuition presented above by building the SOP representation of a data type.

2.2 Building our Universe

An AST consists of a family of data types, with a main one which represents the outer structure of the language, and a number of other possibly mutually recursive data types appearing as arguments to the constructors of the main data type. We can define a very simple language which consists of only one data type, `SExpr` which is isomorphic to binary trees of integers. This choice is just for ease of presentation, the following construction can be applied to a family of mutually recursive data types.

```
data IntTree = Node IntTree IntTree | Leaf Int
```

For each type that appears as an argument to a constructor in our family of data types, we will construct an atom representing that type. Following the example above, we will define

```
data U = KInt | KIntTree
```

Now, we can group all the constructors appearing in our original data types under the `Constr` type in a similar way as we did for the atoms.

```
data Constr :: * where
  CNode :: Constr
  CLeaf :: Constr
```

We have now deconstructed our original data type into two levels: `Constr` corresponding to the level of sums (the constructor) and `U` corresponding to the level of products (the atoms).

Since we had to define two additional data types that have no relation between each other we need a way to tie them together on the type level. To achieve this, we define the `ConstrFor` data type, which can be viewed as a proof that a certain constructor builds element of a certain family. In general an AST consists of a family of possibly recursive data types, hence we will need the additional information about what family the constructor `Constr` belongs to.

```
data ConstrFor :: U -> Constr -> * where
  NodeProof :: ConstrFor KIntTree CNode
  LeafProof :: ConstrFor KIntTree CLeaf
```

Finally we must encode one last bit of information: the "shape" of each constructor. To do so, we can use a closed type family which can be viewed as a function on types. This function takes a `Constr` and returns a list of atoms representing the arguments the constructor accepts.

```
type family TypeOf (c :: Constr) :: [U] where
  TypeOf CNode = '[KIntTree, KIntTree]
  TypeOf CLeaf = '[KInt]
```

We will also need to associate each atom with a singleton, which will allow us to relate terms of our language to their type level representation.

```
data Usingl :: U -> * where
  UInt :: Int -> Usingl KInt
  UIntTree :: IntTree -> Usingl KIntTree
```

Since `TypeOf` returns something of kind `[U]` we will define another GADT named `All`, that maps a type constructo `k -> *` over an argument of kind `[k]` giving us something of kind $*$ to quantify over a list of singletons.

The definition for `All` is straightforward:

```
data All (k -> *) :: [k] -> * where
  An :: All p '[]
  Ac :: p x -> All p xs -> All p (x ': xs)
```

With this setup we can finally construct the `View` data type; this loosely corresponds to a generic view as sum of products of a datatype, and simply deconstructs each term of a type into a constructor and a list of arguments applied to that constructor

```
data View u where
 Tag :: ConstrFor u c -> All Usingl (TypeOf c) -> View u
```

An element of type `View u` represents an element of type `u` deconstructed into its SOP view; the first argument records the choice of the constructor for that datatype and the second is the heterogeneous list of arguments that are required to build that constructor, note that we are dependently relating the shape of the product to the actual choice of constructor (the c).

To conclude the section we can attempt to construct the `View` of a simple expression representing an operation between two values.

```
  expr = Node (Leaf 1) (Leaf 1)
```

The out-most constructor (`Node`) gets mapped to the corresponding `NodeProof`; the shape of the second argument (`[KIntTree, KIntTree]`) is completely determined by this choice.

```
  viewExpr = Tag NodeProof (UIntTree (Leaf 1) `Ac` UIntTree (Leaf 1))
```

Constructing a view can be thought as a way to unwrap the top-most level of the AST; as shown in the example above, the values of the operation are left untouched in the resulting view. Loosely speaking this justifies the intuition that viewing a type as a sum of products gives us information about its shape but does not change any information about its values, enabling us to switch back and forth between representation without losing anything.

3 Type-directed diff

The approach presented by Miraldo and Swierstra [7] takes advantage of the structure encoded in types to define a generic type-directed diff algorithm between typed trees. The inspiration comes from the diff utility present in Unix which is at the heart of the current methodologies employed by VCS

to attempt to compute a patch between two different versions of the same file. The limitations of the diff algorithm, as it currently stands, is that it does not employ any structural information about the data it is trying to calculate a patch on. Files are parsed on a line by line basis and, as the authors show, this is somewhat resilient to vertical changes in the source code. On the other hand it completely breaks down when dealing with horizontal changes, which constitute a heavy chunk of the changes that are usually made to code.

The underlying idea to the approach presented in the article is to employ the generic SOP view presented in the preceding section to define a generic way to view data types. Once that is settled, we obtain a view of any well defined program as a structured tree of data, where each object we inspect can be represented as a choice of a certain constructor, among the available ones, and a choice of arguments to that constructor.

In the process of transforming one tree into another we need to keep track of three different things:

- Changes on the constructor level - The internal nodes of the trees
- Changes on the product level - The branches that go from a node to its children
- Changes on the atoms - The elements that are pointed to by the branches

The atoms can be either other internal nodes, in which case we recurse down the tree, or leaves in which case we record the pair of source and target leaf.

We will use the simple binary tree language presented in previous sections as a working example to show the construction of a patch. The transformation we will walk through is the following: given the following AST:

```
p1 = Node (Leaf 1) (Node (Leaf 1) (Leaf 1))
```

we want to characterize the patch that transforms it to

```
p2 = Node (Leaf  1) (Node (Leaf 2))
```

The first structure we will employ is the *spine*, this can be thought of as a common skeleton between the two trees which captures the parts that do not change under the transformation.

3.1 Spine

We will define Spines to only between two elements of the same sum-type, and defer to later the discussion on how to perform transformation between elements of different sum-types. Calculating a spine for two elements of the same sum-type loosely corresponds to calculating the longest common prefix between two strings. Recall that the two elements x and y are viewed as SOP, in this sense calculating the spine between x and y corresponds to capturing the common co-product structure between them. Let x and y be two such elements, we will have three cases to consider.

- $x = y$
- x and y have the same constructor on the sum level but differ in the arguments
- x and y have different constructors

This gives rise to the following three different constructors for the Spine GADT, each corresponding to one of the cases described above.

```
data Spine (at :: U -> *)(al :: [U] -> [U] -> *) :: U -> * where
  Scp  :: Spine at al u
  Scns :: ConstrFor u s -> All at (TypeOf s) -> Spine at al u
  Schg :: ConstrFor u s -> ConstrFor u r
       -> al (TypeOf s) (TypeOf r)
       -> Spine at al u
```

If the two elements are the same, the spine is a copy. If the top level constructors match, the spine consists of this information and a function to transform the pairs of constructor fields. Lastly, if two constructors don't match, the spine must record this and also contain a function to transform the list of source fields into the list of destination fields.

The `Scp` constructor corresponds to the first case, in which we need to record no additional information other than the fact that the two elements are equal. Before looking at the other two constructors, let us focus our attention for a moment to the three arguments that a spine takes: the third parameter of the spine (the `U`) represents the underlying type for which we are trying to compute a patch, the sum-type to which x and y both belong. The other two, `al` and `at` are respectively a function between products that describes what to do with the different constructor fields and a function

between atoms which describes what to do with the paired fields in case we have the same constructor. These functions are needed for the remaining constructors: the `Scns` constructor corresponds to the case where the constructor is left untouched but some of the arguments have changed. For this reason its second argument consists of the predicate `at` applied to the list of arguments which describes how to transform them.

Finally, `Schg` represents a change of constructor on the sum level: the first two arguments record the source and destination constructors, the third argument is the `al` function applied to the constructor fields of the source and destination constructor respectively.

Let's not worry about the `al` and `at` parameters for the time being, these will later be used to close the recursive knot and generate the full patch by interleaving the construction showed here and in the following sections. For now we can simply observe that if we were to calculate a spine between the two programs introduced above we would proceed by constructing their view as presented in 2.1 obtaining the following

```
p1 = Tag NodeProof (UIntTree (Leaf 1) `Ac` UIntTree (Node (Leaf 1) (Leaf 1)))
p2 = Tag NodeProof (UIntTree (Leaf 1) `Ac` UIntTree (Leaf 2))
```

These views are not completely equal, but their first argument is. This represents the outer choice of constructor and we are after all ultimately transforming an `Node` into another one. The spine produced by these two views will then start with an `Scns` recording the fact that the outer constructor has stayed the same but there are some changes in its arguments.

```
spine = Scns NodeProof _
```

We ignored the second argument up to this point (representing it as an underscore); let's turn our attention to that now: since we know that the constructor is unchanged in the transformation, we also know that both for the source and destination tree, the number and types of its arguments will be the same. For this reason we can simply pair up the corresponding arguments and calculate the diff between every pair. The second argument to `Scns` can be read as: the function `at` applied to a list of pairs of elements of the same type. The type of each pair is specified by `TypeOf s`, which in our working example is equal to $[KIntTree, KIntTree]$. Essentially we have a list of `UIntTree` pairs and are left with the problem of calculating patches between the elements contained in each pair. We can easily see that the first pair of our example will give rise to an `Scp`, the two sub-trees

are in fact the same and we can simply copy the information along the transformation. The second pair is more interesting though: this is the case where we have a change on the constructor level and the spine we produce will be the following

```
Schg NodeProof LeafProof _
```

However the remaining argument to fill is not as simple as the `Scns` case since `Node` and `Leaf` expect completely different arguments. In the case where the constructor had remained the same, we could pair up the arguments and proceed from there; however, when the external constructor has changed, there is no obvious way of pairing up the arguments. Indeed they might be completely in different numbers and types, which motivates the following definition of alignments.

3.2 Alignment

The spine takes care of matching the constructors of two trees, alignments handle the products packed within the constructors. Recall that this alignment has to work between two heterogeneous lists corresponding to the fields associated with two distinct constructors. The approach presented below is inspired by the existing algorithms based on the edit distance between two strings. The problem of finding an alignment of two lists of constructor fields can be viewed as the problem of finding an edit script between them. An edit script is simply a sequence of operations which describes how to change the source list into the destination. In the case of *diff* the source and destination lists are the lists of lines in the source and destination files; in our context the source and destination lists are the products of fields of the source and destination constructors respectively. To compute an edit script we simply traverse the lists, from left to right, considering one element from each list. At each step we are presented with three choices:

- We can match the two elements (Amod) of the list and continue recursively aligning the rest
- We can insert the destination element before the current element in the source list (Ains) and recursively compute an alignment between whatever we have in the source list and the tail of the destination.
- We can delete the element from the source list (Adel) and recursively compute the alignment between the rest of the source and the destination.

The following GADT models the sequence of operations that represent an alignment.

```
data Al (at :: U -> *) :: [U] -> [U] -> * where
  A0   :: Al at '[] '[]
  Ains :: Usingl u -> Al at xs ys -> Al at xs (u ': ys)
  Adel :: Usingl u -> Al at xs ys -> Al at (u ': xs) ys
  Amod :: at u -> Al at xs ys -> Al at (u ': xs) (u ': ys)
```

`Al` is parametrised by the same type-level function introduced in the spine. `A0` represents the empty alignment, `Ains` and `Adel` take as first argument a singleton representing the element being inserted or deleted. These two, together with an alignment for the rest of the list, give us the alignment with an insertion (resp. deletion) as explained in the section above. In the `Amod` case the first argument is the predicate on the underlying atom that describes how to transform it and the second one, as for the case of insertions and deletions, represents an alignment between the rest of the lists.

One key difference to keep in mind, between the edit scripts produced by diff and the alignments is the atomicity of the elements being aligned (lines and sub-trees respectively). In the case of strings we can assume deletions and insertions to be somewhat equivalent in cost thus we can safely try to maximize one of the two. However, in our case, the elements we are inserting or deleting are sub-trees of arbitrary size, therefore it is not obvious if we should try and maximize insertions or deletions.

This poses the problem that when enumerating alignments we have no guiding heuristic to cut the number of solutions, for the time being we will simply ignore the problem and resort to enumerate all possible alignments, avoiding to skew the algorithm into preferring insertions over deletions or vice versa.

Let's walk through calculating the alignment for our running example: we have to produce an alignment between `[KSExpr, KSExpr]` and `[Kint]` (recall that these are the shapes of the `Node` and `Leaf` constructors). Because of the way we define the `Amod` constructor, more precisely because of the `at` function that describes how to transform an atom into the other we restrict ourselves to only attempt an `Amod` between two singletons of the same underlying type. This means that in a case like this one, we will only be able to transform one list into the other by repeated applications of `Ains` or `Adel`. It is worth noticing that this restriction is not mandatory and we could in principle allow these transformations as well, the choice here

is purely pragmatical and the underlying reason is, this will be a recurring theme, to reduce the sheer amount of combinations that must be checked. What we are left with are the three possible alignments that can be formed by inserting the `Uint` and deleting the two `UIntTree`s, we will generate all of them and keep performing the computations on each branch.

3.3 Atoms

Having figured out all the alignments between two lists of constructor fields, we still have to decide what to do in the case where we match two elements. We need to make a distinction between the possibly recursive fields and the constant ones. In the case of constant fields like `Int`s or `String`s, a transformation between two values of this type consists of a pair recording the source value and the destination value. In the case of a recursive datatype we are essentially left with the problem we started from: transforming a value of a data type into another. To do so, we simply start all over again, recursively computing a spine and an alignment between constructor fields. Note that this construction explicitly excludes pairing constant atoms to recursive ones, this choice serves to prune the search space and reduce the number of possible pairings that can be constructed.

To represent pairs of constant atoms we introduce a helper `Contract` datatype which lifts f over a pair of xs.

```
newtype Contract (f :: k -> *) (x :: k) = Contract { unContract :: (f x , f x) }
```

To distinguish between recursive and non recursive elements of the language we define a typeclass with no additional methods, and add instances of this typeclass only for the recursive atoms.

Once again, borrowing the language definition from the previous section, we will have the following class and instances defined

```
class IsRecEl (u :: U) where
instance IsRecEl KSExpr where
```

With this we can define the following datatype to represent diffs between atoms of our language.

```
data At (recP :: U -> *) :: U -> * where
  Ai :: (IsRecEl u) => recP u -> At recP u
  As :: Contract Usingl u -> At recP u
```

Here the `At` datatype is parametrised by a predicate that describes how to transform the recursive atoms. The first constructor, `Ai`, which represents the recursive case is parametrised by this predicate, the constraint is added to ensure by construction that when we build an `Ai` we can only do so for the elements of the language that actually are recursive. The other case is covered by the `As` constructor, recall that in this case `Contract` simply lifts `Usingl` to a pair of elements of type u, so the first parameter can be read as: a pair of `Usingl u`.

In our example we have already seen the case for `Ai`, the atoms paired by the first spine were all `Kexpr`s which we recursively calculated spines on. The `As` will be produced when we match two `Leaf`s that contain different integers, in that case we produce a pair of `Usingl Kint` that record the transformation from one `Int` to the other.

3.4 Recursive alignments

Starting by computing the spine is not necessarily the optimal choice, this can be seen from the following simple example between lists:

```
[ 1, 2, 3, 4 ] -> [ 2, 3, 4 ]
```

Clearly the optimal patch will proceed to delete the first element and then copy over any remaining one. Our definition, however, does not allow for such deletions. Deletions (resp. insertions) are only handled by alignments. To handle such cases we can extend our spines and alignments with the datatype `Almu` that allows insertions or deletions to happen on the sum level.

A match of constructors will be represented as a spine while insertions and deletions will record the constructor being inserted (resp. deleted) and a `Ctx` which records which fields are associated to that constructor. `Ctx`s list zippers [11]: they can be thought as a representation of a type with a hole somewhere; the hole represents the place where we plug in the rest of the tree to continue the computation.

```
data Ctx (r :: U -> *) :: [U] -> * where
  Here :: (IsRecEl u) => r u -> All Usingl l  -> Ctx r (u ': l)
  There :: Usingl u -> Ctx r l -> Ctx r (u ': l)
```

The `Here` constructor represents the hole, or the recursive position in which we want to carry on the computation.

With this definition of contexts, we can finally define `Almu u v`, the datatype that represents structured patches between `u` and `v`

```
data Almu :: U -> U -> * where
  Alspn :: Spine (At AlmuH) (Al (At AlmuH)) u -> Almu u u
  Alins :: ConstrFor v s -> Ctx (AtmuPos u) (TypeOf s) -> Almu u v
  Aldel :: ConstrFor u s -> Ctx (AtmuNeg v) (TypeOf s) -> Almu u v
```

The `AlmuH`, `AtmuPos` and `AtmuNeg` are wrappers around `Almu` s to make source and destination types line up correctly.
This gives rise to another occasion for non-determinism, as `Almu` has the same shortcomings of `Al` meaning that we have no obvious choice of what operation should be maximized over the others. As for alignments we decide to proceed non-deterministically and compute every possibly choice, the next section will introduce some possible optimisations that can be carried out on these two levels to reduce the number of branches being generated at each step.

3.5 Putting everything together

We can write this function from the "bottom up", starting from the atoms and working our way up through spines and recursive alignments. Notice how all these functions return a list of results, as non-determinism comes into play at every step. The signatures have been slightly simplified from the actual implementation where, for example, the result is parametrised by a monad, making it more general. The diff function for atoms should have the following signature

```
diffAt ::  (forall r . IsRecEl r => Usingl r -> Usingl r -> [rec r])
        -> Usingl a -> Usingl a -> [At rec a]
```

This function is parametrised by a function that describes the treatment for recursive atoms. By inspecting the first singleton we learn whether the atom is recursive or not; if that is the case, the function that deals with the recursive elements can be used to build the corresponding `At`. In the other case, when the element is non recursive, we can simply pair up the two constant atoms with a `Contract` and build the non recursive `At`.

We can then proceed by implementing the function that computes all the spines between recursive elements.

```
diffS :: IsRecEl a => (forall r . IsRecEl r => Usingl r -> Usingl r -> [rec r])
      -> Usingl a -> Usingl a -> [Spine (At rec) (Al (At rec)) a]
```

`diffS` lifts the parameter it takes, a function to handle the recursive elements of our language, over the predicate parameters to the spine that results from the two singletons.

We finally have to define a function that computes the diff in terms of `Almu`s. This will call `diffS` in case of two matching constructors from which we can compute the spine wrapping that with the corresponding `Alspn` constructor. In the other cases it will attempt the insertion (resp. deletion) at the constructor level by recording the constructor being inserted (deleted) and producing a `Ctx` which describes where the original tree is attached in respect to the added (deleted) constructor. This function will have the following signature

```
diffAlmu :: (IsRecEl u, IsRecEl v) => Usingl u -> Usingl v -> [Almu u v]
```

As is the case for the alignment between products, here we will simply proceed by enumerating all possible recursive alignments, attempting at each level the alignment of spines, insertions and deletions. One shortcoming with this approach lies in the great combinatorial explosion of possibilities that arises in computing the alignments for constructors and products. The following paragraph will describe an we can employ to prune the number of possible alignments. This optimisation will allow the program to run in an acceptable time in real world scenarios, however we are still at least an order of magnitude slower than `diff3`, this points to the necessity of further and more aggressive optimisations that may be explored in future work.

3.6 Applying Patches

Now that we have constructed these type-safe patches we can define how to apply them to an expression to produce a transformed expression.

Application will be defined between a patch of type `Almu u v` and a singleton `Usingl u`. Again, we will proceed defining our functions from the atoms all the way up to recursive alignments. As before, our functions will be parametrised by one or more functions to deal with the recursive elements.

```
applyAt :: (IsRecEl a => rec a -> Usingl a -> Maybe (Usingl a))
        -> At rec a -> Usingl a -> Maybe (Usingl a)
applyAt appRec (Ai r) x = appRec r x
applyAt appRec (As c) x = if old == new then pure x
                          else if old == x then pure new
                          else Nothing
    where (old, new) = unContract c
```

If we are dealing with a recursive element we can apply the supplied function and proceed the recursive application with that. If the element

is non-recursive, we check if it is a copy, in which we can return whatever argument we got. If it is a change instead, we will check if the argument matches the source and return the target.

Alignments will be applied to heterogeneous list of `Usingl u`. The function is parametrised by the previous function we defined over atoms.

```
applyAl :: (forall a . at a -> Usingl a -> Maybe (Usingl a))
        -> Al at p1 p2 -> All Usingl p1 -> Maybe (All Usingl p2)
applyAl appAt A0 An
  = pure An
applyAl appAt (Amod p a) (Ac x xs)
  = Ac <$> appAt p x <*> applyAl appAt a xs
applyAl appAt (Ains k a) xs
  = Ac <$> pure k <*> applyAl appAt a xs
applyAl appAt (Adel k a) (Ac x xs) = do
  Refl <- testEquality x k
  applyAl appAt a xs
```

Applying an alignment is straightforward, the types guarantee that we can only apply lists of `Usingl`s and alignments which are compatible. This is reflected by the fact that the supplied alignment has type `Al at p1 p2` which matches the type in `All Usingl p1` and the result produced by applyAl has type `All Usingl p2`. We step through the alignment applying each `Al` to the head of the list until we are done.

```
  applyS :: IsRecEl r => (forall a . at a -> Usingl a -> Maybe (Usingl a))
       -> (forall p1 p2 . al p1 p2 -> All Usingl p1 -> Maybe (All Usingl p2))
       -> Spine at al r
       -> Usingl r
       -> Maybe (Usingl r)
applyS appAt appAl Scp x = pure x
applyS appAt appAl (Schg i j p) x = case view x of
  Tag c d -> do
    Refl <- testEquality c i
    inj j <$> appAl p d
applyS appAt appAl (Scns i p) x = case view x of
  Tag c d -> do
    Refl <- testEquality c i
    inj i <$> sAll appAt p d
```

Application for spine is parametrised by a function to apply atoms and one to apply alignments. We inspect the spine and proceed accordingly. If

it is an Scp we return the argument, if it is an Schg we need to inspect the argument and convert it to the SOP view. If the constructor on the sum level matches the source constructor of the Schg then we can construct the element with the target constructor and the result of the application on the alignment. Finally, if the constructor is an Scns, we start by peeling the argument and turning it into its SOP view. If the constructors match, then we can construct the element with the old constructor plus the result of applying appAt to every pair of atoms found by pairing the product in the Scns and the view of x.

Finally, for the case of recursive elements, we can give the following implementation.

```
applyAlmu :: (IsRecEl u, IsRecEl v) => Almu u v -> Usingl u -> Maybe (Usingl v)
applyAlmu (Alspn s) x = applyS (applyAt applyAlmu) (applyAl (applyAt applyAlmu)) s x
applyAlmu (Alins constr ctx) x = inj constr <\$> ctxIns ctx x
applyAlmu (Aldel constr ctx) x = case view x of
  (Tag c1 p1) -> do
    Refl <- testEquality constr c1
    ctxDel ctx p1
```

The case of Alspn amounts to simply calling the function we have defined to apply spines with the correct arguments. In case of an Alins, we still have to construct an element, it will be obtain by an injection of the inserted constructor and the context. ctxIns walks through the context collecting all the singletons and calling the application recursively when it finds the hole. In case of a deletion, we don't have to build anything. We check if the element matches the constructr we intend to delete and simply carry on if it does. ctxDel walks through the context, throwing away every singleton it finds and only calling the recursive application once it finds the corresponding hole.

3.7 Disjointedness

We can define a predicate to decide disjointedness between patches. We want to define this notion only for pair of patches that share the same source, as patches from completely different sources are incomparable to each other. The idea that we want to capture with disjointedness is that two disjoint patches should always commute; this means that we can apply them in any order to the source and always get the same result.

We can start by attempting to define what disjointedness is on the recursive level. Disjointedness should model the fact that two patches are

acting on different parts of the source. This suggests we want to impose the condition that any patch different from the trivial one, is disjoint from itself. Following this line of though we can start by defining

```
disjointAlmu _ _ (Alins _ _) (Alins _ _)
  = False
disjointAlmu _ _ (Aldel _ _) (Aldel _ _)
  = False
```

When matching an `Alins` with anything else, we extract the focus from the context, and recursively call the disjointedness predicate on the focus and whatever the other argument is. In other words, insertions are always allowed.

```
disjointAlmu (Alins constr ctx) almu
  = disjointFromCtxPos ctx almu
disjointAlmu almu (Alins constr ctx)
  = disjointFromCtxPos ctx almu
```

When we match an `Aldel` with an `Alspn` we will inspect the spine contained in the `Alspn`.
If it is an `Scp` then the two are trivially disjoint.

```
disjointAlmu _ _ (Aldel c ctx) (Alspn Scp)
  = True
disjointAlmu _ _ (Alspn Scp) (Aldel c ctx)
  = True
```

If it is an `Scns` then they are disjoint if the recursive changes within the `Scns` do not change the deleted context and the focus of the context is disjoint from the corresponding changes in the `Scns` product.

```
disjointAlmu  (Aldel c ctx) (Alspn (Scns c' ats))
  = case testEquality c c' of
      Just Refl -> disjointFromCtxNeg ctx ats
      Nothing   -> False
disjointAlmu (Alspn (Scns c' ats)) (Aldel c ctx)
  = case testEquality c c' of
      Just Refl -> disjointFromCtxNeg ctx ats
      Nothing   -> False
```

If it is an Schg then they are not disjoint.

The only case left is when we have two Alspn. In this case we have to look at the pair of spines to decide if they are disjoint. As before, we can step through all the cases.

Scp is disjoint from any other node.

```
disjointS Scp s'
  = True
disjointS s'  Scp
  = True
```

A pair of Scns is disjoint if the constructor they fix is the same and if their fields are pairwise disjoint.

```
disjointS (Scns c p) (Scns c' p')
  = case testEquality c c' of
          Just Refl -> disjAts p p'
          Nothing   -> False
```

A pair of Schg is never disjoint.

Finally, when we have an Scns and a Schg, they are disjoint if the constructor fixed by Scns is the constructor changed by Schg and if the fields of Scns are disjoint from the alignment in Schg.

```
disjointS disjointAt (Scns c p) (Schg i j p')
 = case testEquality c i of
         Just Refl -> disjAtAl p p'
         Nothing   -> False

disjointS disjointAt (Schg i j p') (Scns c p)
 = case testEquality c i of
         Just Refl -> disjAtAl p p'
         Nothing   -> False
```

The predicate disjAtAl follows the same pattern outlined up to this point. Since we learned that the constructor of Scns and the source constructor of Schg are the same we know that this alignment has as source exactly this list of fields. We can step through the elements considering them in pairs. Insertions are always fine as long as the rest of the patches are disjoint. If we find an Adel we have to check that the patch on the field being deleted is actually and identity patch. Lastly, when the alignment

contains an `Amod` we can check if the argument of the `Amod` is disjoint from the field.
To do so we need to introduce a function that tells us when two atoms are disjoint, the recursive case can be dealt with a function which will be taken as the first argument to close the recursive loop. Finally, two non-recursive atoms are disjoint if either one of them is the identity (represented by a pair containing the same element).

```
disjointAt :: (IsRecEl a => rec1 a -> rec2 a -> Bool)
           -> At rec1 a -> At rec2 a -> Bool
disjointAt disjointR (Ai r) (Ai r') = disjointR r r'
disjointAt disjointR (As p) (As p')
  = old == new || old' == new'
  where
    (old, new) = unContract p
    (old', new') = unContract p'
```

3.8 Visualization

The patch objects produced by the algorithm are isomoprhic to trees. For this reason, inspecting them by hand is ofter slow and error-prone. To make the inspection of these objects easier we wrote a simple visualiser. It relies on treantJS [12], a library for the creation and manipulation of tree structures. To take advantage of the library we only have to define `ToJSON` instances for our patch type and - depending on how we want to represent insertions and deletions in the tree - possibly for each type in the family that constitutes our language.

Deletion and insertion nodes (both recursive and in alignments) are color coded to be respectively red and green and this coloration is inherited by child nodes and edges. Patches on non recursive elements are represented as a single non-recurisve element, if the patch was a copy, or the pair of elements in the other case. Finally, nodes are collapsable, and the visualisation defaults to collapse every node that only contains copies amongst its children. This allows us to keep the size of the visualised tree restrained, and allows us to read off the important information contained in a patch with ease.

Figure 1 shows the patch constructed between the expressions in the previous section.

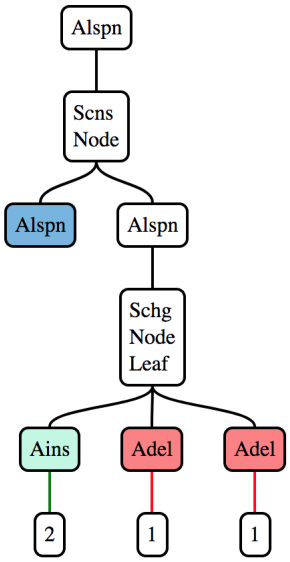


Figure 1: A patch between Binary Trees

3.9 Optimisation

Given the `Al` and `Almu` type defined in the previous paragraphs we are still left with the problem to computing these two levels of alignments. Since we don't know a priori which alignment is more efficient we will non-deterministically compute all the possible ones. The number of all possible alignments can grow very quickly and, to make things worse, we are dealing with alignments of arbitrarily large subtrees, which prevents us from optimising towards insertions or deletions. It is easy to see that in some cases, prioritising deletions can be more profitable and in other it may be better to do the opposite; this uncertainty stems from the fact that at the time we are calculating the alignment we have no information about the size of the sub-trees we are considering.

Despite this limitation, there is one optimisation we can introduce which is to avoid performing insertions if the last step we performed was a deletion and vice-versa. To implement this we must add a parameter that tracks the operation that was taken at the previous steps, we will call this the `Phase`. We can define a function with the same signature as `align` but parametrised with a `Phase`. At each step we inspect the `Phase` and if the last step was

an insertion (resp. deletion) we will only proceed with a match and another insertion (resp. deletion).

The optimisation can be used on the two different levels; the same idea is used to prune the search space for both the alignment between the list of atoms and the recursive alignment between constructors.

4 Clojure

Having developed a general framework to compute patches between typed trees, we know want explore its performance in the context of a real programming language. To test this, we developed a parser for Clojure; the implementation of this parser can be somewhat different to one designed to interpret and run Clojure code. While it may be fruitful to encode as much domain specific knowledge into the parser, thus enabling possible further optimisations. One further consideration is that the parser should try to capture as much syntactical information as possible, in order to produce code that strives to respect any syntactical convention embraced by the authors.

The AST will be composed by a family of data-types, with the `Expr` type representing the "entry point" for each parse.

```
data Expr = Special FormTy Expr
          | Dispatch Expr
          | Collection CollType SepExprList
          | Term Term
          | Comment String
          | Seq Expr Expr
          | Empty

data SepExprList =
   Nil
 | Cons Expr Sep SepExprList

data Term = TaggedString Tag String

data Sep = Space | Comma | NewLine | SEmpty
data FormTy = Quote | SQuote | UnQuote | DeRef
data CollType = Vec | Set | Parens
data Tag = String | Metadata | Var
```

This is enough to parse all Clojure code obtained from the test data that we collected; it can actually parse even more than legal Clojure as it does not consider the semantical correctness of the parsed code, only its syntactical coherence. The definition of `SepExprList` can represent lists of expressions separated by either newlines, commas or spaces, which are all legal legal and interchangeable separators in Clojure, we also have an `SEmpty` separator for the `Nil` case.

We can apply the same procedure outlined before to generate the required singletons and type families. The only difference from before is that in this case is that our language is represented by a family of mutually recursive datatypes. If you recall from section 2 we introduced `ConstrFor` to relate our singleton constructors to the type they were constructing. In case of a single data-type representing the whole AST there was no real necessity for this type, and we could have resorted so simply passing `Constr` around. In the case of the Clojure AST we need this additional information, and the `ConsrFor` type becomes (slightly) more interesting.

```
data ConstrFor :: U -> Constr -> * where
  NilProof :: ConstrFor KSepExprList Nil
  ConsProof :: ConstrFor KSepExprList Cons

  SpecialProof :: ConstrFor KExpr Special
  DispatchProof :: ConstrFor KExpr Dispatch
  CollectionProof :: ConstrFor KExpr Collection
  TermProof :: ConstrFor KExpr Term
  CommentProof :: ConstrFor KExpr Comment
  SeqProof :: ConstrFor KExpr Seq
  EmptyProof :: ConstrFor KExpr Empty

  [...]
```

As the constructor name suggests, we can think of these as proofs that that a certain `Constr` maps to a specific `U`. These are straightforward to generate manually, but can also be generically derived.

5 Heuristics

When moving to a more complex language and real-world data to test on, the specification proposed up to this point, even with the optimisation described in 3.9 we can still only handle very small inputs in an acceptable

time.
In this section we will define some heuristics we can use to guide the process of generating patches in order to trim down the amount of computations that need to be carried on.
These heuristics fall under two categories.
One is to explore the possibility of using the standard unix `diff3` algorithm as an oracle to prune the alignment trees that are being generated. The idea is that instead of enumerating all possible alignments between two trees, we can check and see how *diff*3 treats the sequence of lines in which that tree resides in the source. This can allow us to prune the search space based on the information we can derive from *diff*3 and may be able to speed up the computation to handle larger inputs. This idea can be taken a step further. We can generalise the approach to add an "Oracle" which, based on some internal state, generates the next branches that should be explored.
The optimisation described in 3.9 can be thought of as an oracle too: this oracle has access to the history of choices that were taken on each branch and, at each step it will inspect this history and output only the branches that do not introduce duplication. One of the upsides of this approach is that it will allow us to test different kinds of optimisations in a clean and flexible way; we could even imagine an oracle that interacts with the user, occasionally asking her for guidance into which branches to pursue. Furthermore, we could define a notion of composition between oracles that will give us the chance to combine different optimisations into one.
Another approach that we could take in the attempt to speed up the algorithm is to define an heuristic to score patches which will allow us to greedily prune the search space. With this approach, the question that arises is: what are the properties of patches for which we can compare and score them? The answer is not clear yet. Informally we want to prefer patches that make minimal modifications and encourage copying as much as possible. This is because if a patch consists of a copy on a certain sub-tree, we can be sure that we can safely merge this with any patch that modifies that same sub-tree. In other words, want to define an heuristic that picks the patch that maximises the chances of it being disjoint from any other patch from the same source.

5.1 Basic Oracles

The goal for Oracles is to be able to have a uniform interface to implement different kinds of optimisations, heuristic and possibly even human

interaction in the process of generating all the possible patches.

The key idea, is to extend the algorithm to perform a monadic action at each non-deterministic "junction". The result of this action will be a list that encodes which branches should be explored and which should be cut from the enumeration of patches.

We can start by observing that in both places where we have non-determinism, we always have a choice between three possible paths: to insert/delete something, or to try and modify a pair. We can model this with a very simple datatype

```
data Path = I | M | D
```

Where the three constructors respectively stand for: Insert, Modify and Delete. As the optimisation described in the previous section requires knowledge about which path we took at the previous step, we want to give our oracles the possibility to inspect the history of issued paths on each branch.

We can now define our Oracle class

```
type HistoryM = ReaderT [Path]

class Oracle o m where
  callP :: o -> All Usingl p1 -> All Usingl p2 -> HistoryM m [Path]
  callF :: (IsRecEl u, IsRecEl v)
        => o -> Usingl u -> Usingl v -> HistoryM m [Path]
```

The Oracle class has two functions, one for the choice on the constructor level and one for the choice on the product level. At each choice, the oracle has access to the history of paths issued on the branch via a reader monad, and has also access to it's internal state (the *o* type). Notice that the signatures differ in the arguments they take, `callP` is meant to handle alignments on the product level, as such it takes the two heterogeneous lists that are being aligned. `callF` on the other hand, is meant to handle the recursive alignments on the constructor level and takes two singletons as input (the 'F' stands for fixpoint)

5.1.1 NoOracle

To get back our original behaviour we can define the most simple Oracle that simulates the non-deterministic choice:

```
data NoOracle = NoOracle
instance (Monad m) => Oracle NoOracle m where
  callP _ An         An         = return []
  callP _ An         (_ `Ac` _) = return [I]
  callP _ (_ `Ac` _) An         = return [D]
  callP _ _          _          = return [I , M , D]

  callF _ _ _ = return [I , M , D]
```

This oracle will not contain any global information and will ignore the history of issued paths. It will simply output all possible choices in any non-trivial case.

5.1.2 NoDupBranches

We can encode the same optimisation presented in 3.9 as an oracle. We can define the following function that looks at the history of issued paths to avoid performing an insertion if the last step was a deletion and vice-versa.

```
nextPaths :: [Path] -> [Path]
nextPaths (I:_)     = [I, M]
nextPaths (D:_)     = [D, M]
nextPaths (M:_)     = [I, M, D]
```

With this function we can implement the `NoDupBranches` oracle

```
data NoDupBranches = NoDupBranches

instance (Monad m) => Oracle NoDupBranches m where
  callP _ An         An         = return []
  callP _ An         (_ `Ac` _) = return [I]
  callP _ (_ `Ac` _) An         = return [D]
  callP _ (s `Ac` _) (d `Ac` _) = ask >>= return . nextPaths

  callF _ s d = ask >>= return . nextPaths
```

5.2 Oracle composition

With the oracles we gain the possiblity to tweak the run-time behaviour of the algorithm without have to change any parts of the actual implementation. An advantage of this is that we can define a notion of composition

between oracles, this way, we can layer different processes each of which is independent of the other in terms of implementation.

The composition we define wants to model a stack of oracles, the oracles are interrogated in the order in which they appear on the stack. When an oracle is interrogated, only if the answer is an empty list we will go down the stack and ask the oracle underneath.

This means that we can build other optimisations on top of `NoDupBranches`, these optimisations can also be partial or based on heuristics, as long as we have a "safe" oracle at the bottom of the stack we can always fallback to the ones below in the cases when it is not clear which choice should be done.

```
  data ComposeOracle a b = ComposeOracle a b

-- Give it a nice constructor
(<°>) :: a -> b -> ComposeOracle a b
a <°> b = ComposeOracle a b

instance (Monad m, Oracle a m, Oracle b m) => Oracle (ComposeOracle a b) m where
  callF (ComposeOracle a b) s d = do
    o1 <- callF a s d
    case o1 of
      [] -> callF b s d
      o1 -> return o1

  callP (ComposeOracle a b) s d = do
    o1 <- callP a s d
    case o1 of
      [] -> callP b s d
      o1 -> return o1
```

5.3 DiffOracle

A considerable speedup can be obtained by using *diff* to prune the search space. We can define a datatype, which mirrors the definition of a path and identifies which lines are copied, which are deleted and which are inserted according to *diff*.

We will call this datatype `DiffAction` and it will have the following definition.

```
  data DiffAction =
    OMod LineRange LineRange
```

```
  | OIns LineRange
  | ODel LineRange
```

Where `LineRange` is a pair of `Int` s which represent the first and last line of the region. We will produce an OMod every time we have two contiguous regions of insertions or deletions between the source and destination file. For example, given this pair of source and destination

```
(defn function
  [a b]
  return a)
```

```
;; A comment
(defn function
  [a b]
  doSomethingElse
  return b)
```

The preprocessing will produce the list [OIns (1,1), OMod (3,3) (4,5)]. The first line can be definitely ruled as an insertion as it is adjacent to a copy on both the source and destination. *diff* records line 3 of the source as a deletion and lines 4 and 5 of the destination as insertions. This gives us two contiguous regions of insertions and deletions between source and destination and are accordingly recorded as a Mod.

Since traversing the whole list of `DiffAction` every time we want to call the oracle is not very efficient we can encode the same information in a pair of maps from `Int` to `Path` (one for the source and one for the destination files).

Given this pair of maps, the oracle will extract the `Linerange` contained in each `Usingl u` and lookup the line that corresponds to the beginning of the `LineRange` in the corresponding map.

We can define the following helper predicate

```
isMod :: LineRange -> M.IntMap Path -> Bool
isMod lr m = case M.lookup (takeStart lr) m of
  Just M -> True
  _      -> False
```

and the corresponding `isIns` and `isDel` we can finally define the `giveAdvice` function.

```
giveAdvice :: DelInsMap -> Usingl u -> Usingl v -> [Path]
giveAdvice (srcMap, dstMap) src dst =
  if (isMod srcRange srcMap && isMod dstRange dstMap)
    then []
```

```
  else if (isDel srcRange srcMap || isMod srcRange srcMap)
    then [ D ]
  else if (isIns dstRange dstMap || isMod dstRange dstMap)
    then [ I ]
  else [ M ]
  where
    srcRange = fromJust $ extractRange src
    dstRange = fromJust $ extractRange dst
```

In case in which the ranges do not match with anything in our maps, we simply want to emit an `M`, as these expressions lie in lines that diff3 identified as copies (we will see that this gives rise to some problems as *diff* can copy in a way that our algorithm does not support). When both source and destination ranges match with the ranges of a `Mod`, we are essentially in a range where we know that *diff* can not reconcile the changes between source and destination and will simply delete everything in the source and insert everything in the destination. This is the case where we can attempt to generate a more efficient patch, so we return an empty list to fall back to any underlying oracle that will compute a full enumeration in that range. We want to add the two other else branches to handle the case when transforming a source into the destination, we end up on a branch that has inserted (resp. deleted) everything it had to, and it simply needs to delete (rep. insert) the remaining range.

Finally, the full oracle can be defined as follows:

```
instance (Monad m) => Oracle DiffOracle m where
  callF o s d = return \$ askOracle o s d

  callP _ An         An         = do
    return []
  callP _ An         (_ `Ac` _) = do
    return [ I ]
  callP _ (_ `Ac` _) An         = do
    return [ D ]
  callP o (s `Ac` _) (d `Ac` _) = return \$ askOracle o s d

askOracle :: DiffOracle -> Usingl u -> Usingl v -> [Path]
askOracle (DiffOracle diffActions) src dst = case (extractRange src, extractRange dst
      (Nothing, Nothing)         -> [ M ]
      (Just sRange, Nothing)     -> [ D ]
```

```
(Nothing, Just dRange)     -> [ I ]
(Just sRange, Just dRange) -> giveAdvice diffActions src dst
```

We can exploit the fact that the `LineRange` is only defined for the recursive elements of the family, which means that in the case we can not extract it, we can short-circuit the computation since we never pair up non-recursive elements with recursive ones.

5.3.1 Shortcomings

One of the issues with this optimisation is that not every modification between source and destination file is correctly identified by this procedure. Some problems arise when changes in the resulting AST are invisible to diff. Suppose that we didn't include empty expressions in our AST and modeled the top-level parse as either a single expression or a sequence of multiple ones. What happens when try to compute the patch between the following files?

```
1 (keep
2   old keep)
```

```
1 (keep
2   new keep)
3 (new new)
```

The preprocessing will produce the following list of `DiffAction`: `[OMod (2,2) (2,3)]`. However, to perform the optimal patch between these two programs we clearly want to insert a `Seq` on the first line and proceed with the diffing from there. By following the *diff* instead, we realize we had to change the external node when it is too late, we are already on line 2 and have already decided to copy over the first line. At its core, the problem lies in the fact that the external change is invisible to *diff*, since it only deals with lines, and the addition of lines at the end does not influence in any way the lines at the beginning of the file. In our case, we have the whole expression tree, and adding a line at the end of the file does not necessarily only trigger changes on the leaves of the tree, but changes can bubble up to the root, as in the example shown.

Possible Solutions

There are different ways we can solve this problem. One is to carefully think about which nodes have this "non-local influence" and treat them in a special way. For example, the problem presented in the previous snippet arises from the fact that the top-level the parser is designed to parse either an `Expr` or a `Seq` of expressions, but we can only detect this change when we get to the second `Expr`.

Whenever we change a single expression to a sequence of two by adding one at the end, we will record the insertion, but will also mark the first line as a copy. This means it will be "to late", when we get to the inserted line, and we already decided to copy a node which leaves us no slot to insert the new expression.

One solution is to design the AST in a way that prevents the problem from ever arising. If we modelled the top-level of a program, so that the top level is always a `Seq` and we add an `Empty` constructor to `Expr`, then an example like the previous one would not be problematic anymore. Indeed, when we get to the second line, we don't have to change the previous constructor anymore, as the source will conveniently contain an `Empty` expression slot in which we can insert our new expression. This solution has the problem of being very ad-hoc. We have to modify the parser and add support for the new constructors in our generic module. Empty expressions turned out to be useful in any case, as we need them to model the patch that inserts into an empty file, but the solution is clearly not satisfactory as it requires us to model our AST in a very specific way to avoid it.

A better solution would be to design oracles that can deal with the problematic cases. Sticking to the example presented earlier, we could define a simple oracle which ignores everything, except the case when it gets as input a single expression from the source and a sequence of multiple in the destination. In that case we know that, no matter what the other oracles say, we must insert a `Seq` node on the top-level, we can directly return `I` and skip the other oracles.
This solution is better than the previous one, as it is less invasive. It is still troublesome however that the correctness of the oracle depends on an oracle being present before it. Also, if we want to add support for other languages in the future we will have to carefully re-implement the ad-hoc oracles for the new languages. To solve these last lingering issues, we can adopt a different strategy.

Ultimately the problem lies in the fact that there are two distinct actors at play: on one hand we have the AST, on the other the actual lines of code, which can be thought of as a pretty-printed representation of the AST. We are using the line based solution over the pretty-print of the AST to derive a solution on the AST. Our problems come from the fact that changes to the pretty-printed AST don't always map one-to-one to changes in the AST. If we could somehow run the line based diff on the AST directly there would

be no discrepancies. To do so, we can print the AST to a file and, since every constructor is annotated with a line range which tell us on what line it appears, we can print each sub-tree on the lines the elements it contains appear in. We can now run *diff* to preprocess the ASTs printed in this way. Each change that was detected by running on the original files will also be detected by looking at the ASTs, as they are representations of the content of those files. The converse, is not true though, by running *diff* on the AST we know exactly on which levels of the tree, modifications on certain lines reflect. The problematic case we showed earlier will now be correctly recognized. Indeed, by printing the ASTs of the two files showed earlier, we will immediately detect that on line 1 the constructor has changed from `Collection` to `Seq`.

5.3.2 Further Shortcomings

Another problem, perhaps even more troubling, is that *diff* can mark some parts as copies even though these copies are illegal for our algorithm. Imagine a case where we have this pair of source and destination files

```
1 ((keep
2   del
3 keep))
```

```
1 ((keep
2   ins)(
3 keep))
```

In this case, according to `diff3` we are supposed to copy lines 1 and 3 and modify line 2. The problem is that - in the source file - we have a `Cons` node that contains all the copied lines in one of its children. In the destination however, some of these lines are copied to the left child of a Cons, and others to the right. What happens is that *diff*3 can copy nodes across adjacent sub-trees, but our algorithm does not, we have to pick one of the two sub-trees and attempt the copies only into that.
When we are transforming a `Cons a Nil` into a `Cons c d` we only have the option of inserting d in the place of Nil. However, according to *diff*, lines in the sub-tree `a` get copied both to `c` and `d`. In practice this means that when we are at the step in which we have to calculate the patch between a and c, we are left with inconsistent information, in particular we marked some nodes as copies into `d` when we actually want to delete them.

Possible Solutions

To solve this problem we must find a way of detecting when a situation like the one described above arises. We can formulate the problem as follows:

given the following source and destination nodes

```
src = Seq a b
dst = Seq c d
```

- Let $CopyPairs(P)$ be the set of pairs (s, t) that are marked as copies by *diff*3 in the program P.
- Let $CopySet(e)$ be the function that extracts all the lines marked as copies contained in an expression e (A line is marked as copy if it appears on the left in one of the pairs resulting from $CopyPairs(P)$).

Then, if we can find an $s \in CopySet(a)$ such that $t \in CopySet(c)$, where $(s, t) \in CopyPairs(P)$ and we can find $s' \in CopySet(a)$ such that $t' \in CopySet(d)$, where $(s', t') \in CopyPairs(P)$ then we know we are in a conflicting situation.
This condition deals with the case where we had to insert a sub-tree which contains at least one expression marked as copy, to deal with the deletion as well, we must also check that the converse condition does not hold going from c to a and b.
Once we detected the conflicting copy across sub-trees, we can simply remove pairs from the set $CopyPairs(P)$ until the previous condition is satisfied. In other words, after we detect that a node contains copies over sub-trees, we pick one of the sub-trees and remove all the copies contained in it.

The only part that is missing is about which criteria can be used to pair up the nodes we must check. One way to formulate the problem is the following: we run into this issue whenever we have a pair of candidate nodes that are marked as copy on their outer level (resulting in an `Scns` in our case), but would then later attempt to copy across subtrees. The crucial observation is that the candidate nodes to check will always start on a line that *diff* will have marked as copies (this is exactly the issue, *diff* can issue a copy there and move across sub-trees later, our algorithm can not).

One solution could be to extend the oracles to add support for a `De-optimise` operation. The oracle should be able to perform some computation on each branch and decide if it should keep operating on subsequent computations that follow from that branch. This could very well consist in performing the check we described earlier to detect when the information contained in the DiffOracle is not trustable anymore, and proceed the process ignoring the oracle which is marked as de-opt. Supporting this interface might be also

very useful if we plan to write a human-interaction oracle, where we could have it work on the top levels of the tree (e.g. user matches top level expressions) and then turn off as the program takes care of the of the deeper levels.

Another solution, could be to try and solve the problem with some preprocessing.
This criteria identified to match nodes suggests an enumeration strategy that we could use to break any eventual conflicts in the `[DiffAction]`. The idea is to identify which lines *diff* marked as copies and collect all the subtrees on those lines (from the source and destination respectively) so that we can perform the check on them. The final observation we need is that, when collecting sub-trees starting on a pair of lines (s, r) - where s is a copied line from the source file that ends up on line d in the destination - then we will collect the same number of sub-trees from s and d. This happens because since the line is a copy, it must contain the same number of sub-trees starting from it, both in the source and destination file.
Moreover, since the whole line is marked as copy, our algorithm will end up only considering the respective pairs of sub-trees, meaning that we can zip up the two lists and check each pair.
Given, a function `CollectST(P,s)` that takes as input a program `P` and collects all sub-trees starting on line `s` in `P`.
For each pair $(s, t) \in CopySet(P)$ we can perform the following steps:

```
let src = CollectST(P,s)
let dst = CollectST(P,t)
map (check src dst) (zip src dst)
```

Where `check` is the predicate we defined earlier to check if two expressions are conflicting. At this point, if check fails, we have a pair `(s,t)` of lines which *diff* identified as copies but our algorithm can not. Once identified the critical pairs of lines we can remove pairs from $CopySet(P)$ until the predicate is true.

5.4 Cost

Up to this point we have seen how the algorithm generates a list of patches, we have explored some techniques to reduce the size of this output list but we still have never considered the problem of which patch choosing from the output.
We can consider patches as partial functions; in this context, an application function is more accurate than an other if it succeeds in producing a patched

result to more input data.
We can lift the canonical partial order on `Maybe a` to functions by pointwise comparison over their domain. More formally, we obtain

$$\begin{aligned} \text{Nothing} &\leq p \\ \text{Just}\, p1 \leq \text{Just}\, p2 &\iff \forall x.\ p1\, x \leq p2\, x \end{aligned}$$

This definition - while capturing our intuition faithfully - is too extensional to be practical. The approach we will take is to assign a natural number to every patch, representing its cost. In this way we can compute this value by just traversing the patch, ideally this ordering should respect the extensional definition. Intuitively the best patch between x and y is the one that fixes as little elements as possible in the domain and range of its application.
We will count the elements that are fixed by performing insertions, deletions or replacement of non-recursive atoms. With this in mind we can start defining the cost function for trivial patches.

```
costK :: TrivialA u -> Int
costK c = if old == new then 0 else 2
  where (old, new) = unContract c
```

If the patch is a copy then no element is fixed, if it contains a replacement instead we fix one item in the domain and one in the range giving a cost of 2.
In the general case of atoms, we just need to distinguish between recursive and non-recursive elements and call the appropriate function.

```
costAt :: (IsRecEl a => rec a -> Int)
       -> At rec a -> Int
costAt costR (As pair) = costK pair
costAt costR (Ai spmu) = costR spmu
```

With this function we can now define the cost for alignments.

```
costAl :: (forall a . at a -> Int)
       -> Al at p1 p2 -> Int
costAl costAt A0 = 0
costAl costAt (Adel a al) = costUsingl a + costAl costAt al
costAl costAt (Ains a al) = costUsingl a + costAl costAt al
costAl costAt (Amod at al) = costAt at + costAl costAt al
```

Here we need a cost function over `Usingls`, intuitively we want to assign a higher cost to a patch that deletes (resp. inserts) bigger elements. We can assign a cost to each `Usingl` and to each type in the family that constitute our target language by counting the number of choices that are fixed by each element. In the case of BinaryTrees - which have been our running example - this translates to

```
costUsingl :: Usingl u -> Int
costUsingl (UInt u) = 1
costUsingl (UIntTree t) = costIntTree t

costIntTree :: IntTree -> Int
costIntTree (Node t1 t2) = costIntTree t1 + costIntTree t2
costIntTree (Leaf i)     = 1
```

We can now define cost for spines.

```
costS :: (forall a . at a -> Int)
      -> (forall p1 p2 . al p1 p2 -> Int)
      -> Spine at al u -> Int
costS costAt costAl Scp = 0
costS costAt costAl (Scns c p) = sumAll costAt p
costS costAt costAl (Schg i j p) = costAl p
```

The definition for the function `sumAll` is omitted. It simply computes `costAt` over each pair of elements and sums all these together.
Finally, we are only left with the recursive alignments to handle. The case of `Alspn` has been handled already, if we match on an `Alins` or an `Aldel`, then we want to add 1, which represents fixing the choice of the external constructor being inserted or deleted, and the cost of the context.

```
costAlmu :: Almu v u -> Int
costAlmu (Alspn sp) = costS (costAt costAlmuH) (costAl (costAt costAlmuH)) sp
costAlmu (Alins c ctx) = 1 + costCtxPos ctx
costAlmu (Aldel c ctx) = 1 + costCtxNeg ctx
```

The cost of the context, as in the case for alignments, sums the cost of all the `Usingls` in the context with the result of recursively calling `costAlmu` when we reach the hole.

5.5 Bounded Search

The last optimisation we are going to present is very simple but will allow us to gain just enough benefit to be able to handle the majority of our test data. Bounded search is a widespread technique employed when we have an exponential number of solutions to a problem and a *quality* function which assigns a value to any (possibly partial) solution. We can start the search process by establishing an upper bound to the quality of solutions we want to consider.
As we move through the solution space we want to keep track of the current quality of the solution, and prune every branch that exceeds the imposed bound. This process is not guaranteed to generate a solution, the upper bound to the solution quality might be too low. For this reason, bounded search implementations usually have some strategy to restart the search in such cases, increasing the bound by a suitable amount. On the other hand, if the bounded search terminates, we know that the optimal solution (where optimal means minimal according to the quality function) must be in the produced results.
A perfect candidate for the quality function is the notion of cost defined above. However we want to adapt it so that it can work on partial solutions, furthermore we want to be sure to apply the pruning as soon as possible and reduce the number of steps performed in a branch that will be pruned.
We can extend our History monad to store an `Int` which will represent the cost of each branch. Now we just have to change the algorithm to update this local cost at every step and check if the bound hasn't been exceeded at each step.
We can omit the details of how this cost is assigned, as it mirrors exactly the definition of the cost function over patches presented in 5.4.
We will thread the current cost across the computation at each alignment step (recursive and non) we will add the cost of performing that step to the total and, if we exceed the supplied upper bound, prune the branch immediately.

6 Experimentation

6.1 Domain specific improvements

Before presenting the results we want to introduce a relaxation of the definition of disjointedness. Many of the conflicts appearing in the collected test data share a common pattern. These are conflicts that arise from the

equivalent of configuration files, which in Clojure are often expressed in the language itself via the `defproject` macro.
For example:

```
(defproject project-name "1.2.3"
  :description "The description of the project"
  :dependencies [[dependency-1 "0.0.1"]
     [dependency-2 "1.1.0"]]
  :dev-dependencies [[dev-dependency-1 "1.5.1"]])
```

It is very common for these files to give rise to conflicts due to modifications in the required version of the different dependencies by commits which are either merged into or rebased on top of the development branch.
We can define two patches to be structurally-disjoint if they only differ on non-recursive atoms. Given a pair of structurally disjoint patches we can employ some domain specific merging strategies which may automate or drastically reduce the user effort required to perform the merge. In a case like the one described above, most of the conflicts are resolved by picking the highest version number every time there is a conflict consisting of a choice between two strings encoding version numbers. Nowadays, most projects adhere to SEMVER [13] which defines a standard total-ordering for strings representing versions which partially encode the semantics of the change.
This suggests that conflicts arising from structurally-disjoint patches can often be automatically via some user-defined partial ordering between the atoms of the language. For the case of SEMVER specifically we could imagine encoding different specific resolution strategies e.g. we could decide to automatically resolve conflicts between patch and minor version changes and still notify the user when there is a conflict on a major version change.

There is another relaxation of the disjointedness predicate which we may consider. The current definition of disjointedness imposes that every patch p, that is different from the trivial patch `Scp`, is not disjoint from itself. We can attempt to relax the definition from disjointedness to compatibility: two compatible patches may modify the same elements, as long as they do so in the same way. It is easy to see that every patch is compatible with itself.
Because of the current practices in how branches and the development process is managed, it is not so rare that we may need to merge a pair of patches which both contain the same changes. These common changes can come from cherry-picking another commit, rebasing, merging or any similar operation. Compatible patches can still not be applied in commuting order

with the same result unless we relax the definition of application as well. However, it is easy to see that given two patches from the same source, and a proof that they are compatible, we can produce a single patch which encompasses the same changes and is independent of the order in which the two patches are supplied to the function. To put it another way: two patches are compatible if we can combine the into one without throwing away any information and the function that combines them is commutative in its arguments. Clearly disjointedness implies compatibility by picking the merging function to be the application we defined earlier.

6.2 Results

In order to test the framework in a real world context we need to find some suitable data. To acquire this we explored all the Clojure repositories on Github and extracted the ones with the best combination of stars and collaborators. A high number of collaborators will possibly imply a higher chance for conflicts in the source tree, the high number of stars is a good indicator of the quality of the Clojure code and hopefully provides a selection of repository from different domains.
What we need is a way to identify merge points in a projects history; we also want to know how *diff3*performed in those merges, in essence: we want to find all merge points and record if the merge was performed automatically or a conflict had to be manually fixed.
We wrote some scripts to mine the data from Github and to walk through the source trees mining conflict points that can be used to test the framework.
Merge commits, are identified by the simple fact of having more than one parent, we will restrict our attention to the case of two parents. This is the most common and sensible case for our scenario as most teams will develop features on different branches and eventually merge each branch into master. This process will generate a merge commit with exactly two parents, one of which is the master branch and the other of which is the feature branch.
For each of these commits we reproduced the process of making the merge within branches and extracted any Clojure files that were marked as conflicts from this merge.
For each of these files we want to extract three different versions of it. The original version `O.clj` represents the snapshot of the file at the moment the branches initially diverged. It is the last version the two branches agreed on. We also want to extract the versions `A.clj` and `B.clj` which capture the current state of the file on each of the two branches.

From this processing we extracted 135 folders, each containing the three snapshots of the conflicting file. These files have been shrunk with a preprocessing that relies on `diff3` to remove all top level expressions that are are not involved in a conflict in any way.
Each test consists in generating the pair of patches OA and OB and checking if they are disjoint. The patch OA is the best patch generated between `O.clj` and `A.clj` according to the cost function, and OB is generated in the same way but with `B.clj` as destination.
Conflicts were mined from the following repositories

- clj-http
- incanter
- lein-figwheel
- leiningen
- riemann
- ring

The following table shows the results of running the tests for disjointedness and compatibility, both in the full and the structurally-respecting variations.

18 timeout

Predicate	True	False
Disjoint	32	84
Structurally-Disjoint	88	0
Compatible	42	0
Structurally-Compatible	97	19

[Some discussion of specific numbers?]

7 Conclusion

7.1 Related Work

In this thesis we focus on converting the theoretical approach presented in [7] into a practical implementation that can be tested against real-world data. Previous attempts to tackle this problems varied in the approach taken.

Untyped approaches have been extensively studied: with authors focusing both on the linear [**?** **?**] and the tree [**?** **?** **?** **?** **?** **?**] variation.
In recent years other authors explored the typed approach **??** in a generic setting. However they restricted their attention to the linear variation, by considering a flattening of the tree consisting in a pre-order traversal. This flattening however, makes it harder to guarantee that the transformation encoded in a patch is structurally preserving.
Several pieces of related work exist in the literature: from VCS systems built on strong theoretical foundations like Darcs and Pijul, respectively based on work by Roundy [14] and Mimram [**?**]. In our experimentation we have used Git to extract the information required for a merge (e.g. picking the common ancestor between two files). It is interesting to note how O'Connor [**?**] and other authors point out how the strategy adopted by Git for merges is inherently inconsistent and can lead to some surprising outcomes compared to other VCS.
Finally, Swierstra et al. [**?**] showed how separation logic and Hoare calculus can be used to reason about patches, in particular in terms of characterizing the relationship between patches.

References

[1] Lindley, Sam, and Conor McBride. "Hasochism: the pleasure and pain of dependently typed Haskell programming." ACM SIGPLAN Notices 48.12 (2014): 81-92.

[2] Eisenberg, Richard A., and Stephanie Weirich. "Dependently typed programming with singletons." ACM SIGPLAN Notices 47.12 (2013): 117-130.

[3] Yorgey, Brent A., et al. "Giving Haskell a promotion." Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation. ACM, 2012.

[4] Miraldo, Victor Cacciari, and Wouter Swierstra. "Structure-aware version control: A generic approach using Agda." (2017).

[5] Swierstra, Wouter, and Andres Loh. "The semantics of version control." Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. ACM, 2014.

[6] Vassena, Marco. "Generic Diff3 for algebraic datatypes." Proceedings of the 1st International Workshop on Type-Driven Development. ACM, 2016.

[7] Miraldo, Victor Cacciari, Pierre-Évariste Dagand, and Wouter Swierstra. "Type-directed diffing of structured data." Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development. ACM, 2017.

[8] de Vries, Edsko, and Andres Löh. "True sums of products." Proceedings of the 10th ACM SIGPLAN workshop on Generic programming. ACM, 2014.

[9] Hickey, Rich. "The clojure programming language." Proceedings of the 2008 symposium on Dynamic languages. ACM, 2008.

[10] Eisenberg, Richard A. "Dependent types in Haskell: Theory and practice." arXiv preprint arXiv:1610.07978 (2016). APA

[11] Huet, Gérard. "The zipper." Journal of functional programming 7.5 (1997): 549-554.

[12] http://fperucic.github.io/treant-js/

[13] https://semver.org/

[14] Roundy, David. "Darcs: distributed version management in haskell." Proceedings of the 2005 ACM SIGPLAN workshop on Haskell. ACM, 2005.