Report

Giovanni Garufi

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 is called an edit script. Edit scripts are used 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. This operation is clearly not always possible: if two collaborators have modified the same line , in which case the two resulting edit scripts will "overlap", it is not clear which one of the two modifications should be picked.
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 one will be the one we want to pick; however the diff tool makes a big assumption in considering lines to be the basic units in which change is observable.
According to diff, two unrelated changes on the same line would still give rise to a conflict that requires manual intervention. The main idea of this research is to design an alternative to diff which offers a finer grained control over the units of change, approaches similar to this one have already been explored in previous literature [5], [4] and [6] but the novelty in the approach that will be presented relies on making heavy use of type information to guide the algorithm. In particular, in the context of programming languages we already have abstract syntax trees (AST) that encode the semantical structure of a program. By attempting to extend the diff algorithm to operate on the AST as opposed to simple lines of text, we will gain both more control over the units of change gain information about the structure,

encoded in the types, which we can use to create transformation that operate on this structure in a principled way. The drawback of this approach is that it moves the problem from the "flat" world of strings to the "layered" world of trees, and the problem of computing a patch between two elements suddenly becomes much more computationally expensive.

2 Overview

In the remainder of the proposal we will show a full Haskell implementation of the algorithm presented by Swierstra and Miraldo [7], which is presented in Agda in the original paper.

The following section is 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 characterise 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.

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. Despite the generality of the algorithm, in this work, 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.

In the last sections we will review some of the possible future directions that can be explored with this framework both in terms of gathering concrete evidence about the performance of the algorithm and extending it to explore the different design choices that were made along the way.

3 Dependent types in Haskell

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 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 -> Nat`.

We can use the `Nat` kind to index generalised 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 runtime 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 runtime. That information is only used at compile time to check that everything "lines up" the way it should be, but could actually be erased at runtime.

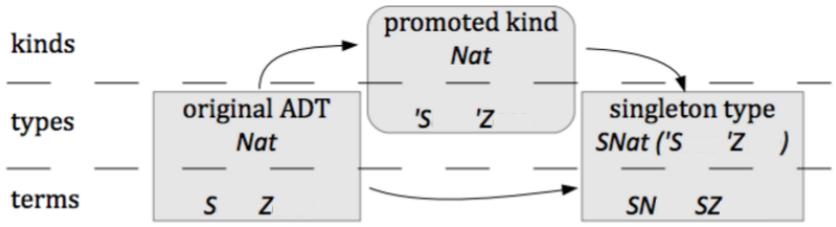
Suppose we want to write a `split` function, this function takes an `n` of kind `Nat`, a vector of length `n :+ m` and splits it into a pair of vectors, with respectively `n` and `m` elements.

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.

The following figure gives a good representation of this process: the DataKinds extensions promotes things of type `Nat` to things of kind `Nat`. The singleton allows us to take one step back in this ladder, and associates to every thing of kind `Nat` a term from the singleton type `SNat`. The following picture, borrowed from [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 runtime decisions based on the types we obtain.

Singletons solve the two problems outlined above: they have kind `*` and it contains 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, [1] shows us how to emulate implicit types via type classes and ultimately shows how all kinds of quantification, modulo some boilerplate, are possible in Haskell.

4 Sum of Products

The basic idea of the SOP universe is to view any term as a choice of a constructor and a list of the argument required for 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 possibly takes a different number of arguments of possibly different types. 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 example). 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 simple 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 once we are done acting on the representation. This encoding is presented by Loh et al. [8] and represents one of the

different ways to approach generic programming in Haskell. The remainder of this section will try to formalise the intuition presented above by building the SOP representation of a simple data type.

An AST consists of a family of datatypes, with a main one which represents the outer structure of the language, and a number of other possibly mutually recursive datatypes appearing as arguments to the constructors of the main datatype. We can define a very simple language which consists of only one datatype, *SEpxr* which is isomorphic to binary trees of integers

```
data SExpr = Operation SExpr SExpr | Value Int
  deriving (Eq, Show)
```

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

```
data U = KInt | KSExpr
```

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

```
data Constr = Operation | Value
```

We have now deconstructed our original datatype into its two corresponding levels: `Constr` corresponds to the level of sums, the constructor and `U` corresponds to the level of products, the atoms.

Since we had to define two additional datatypes 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. Since in general an AST consists of a family of possibly recursive datatypes, here we will need the additional information about what family the constructor `Constr` belongs to.

```
data ConstrFor :: U -> Constr -> * where
  OperationProof :: ConstrFor KSExpr Operation
  ValueProof     :: ConstrFor KSExpr Value
```

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 Operation = '[KSExpr, KSExpr]
  TypeOf Value = '[KInt]
```

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

```
data Usingl :: U -> * where
  Uint    :: Int -> Usingl KInt
  Usexpr  :: SExpr -> Usingl KSExpr
```

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

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` datatype, 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 heterogenous 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).

5 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 between the data it is trying to merge. 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 but completely breaks down when dealing with horizontal changes, which constitute a heavy chunk of the changes that are usually made to code.

Suppose we have the following innocuous looking function in clojure:

```
(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 algorithm we will run into problems. Despite the two are modifying disjoint pieces of the actual code, the diff algorithm, which is employed by most VCS when trying to compute a merge patch between three objects will output a conflict. The reason is that both changes touch the same line, this is both a nuisance in terms of having to manually solve the conflicts, but more importantly also introduces non determinism, as the presence of conflicts may depend on things like indentation instead of being a fundamental property of the transformation.

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 datatypes. 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. This process is equivalent to parsing the source language into an AST, an interesting consideration to make is that the choice of AST for diffing purposes might be very different from the representation that would be chosen for a compiler of the language. While these two structures should be "somewhat isomorphic", the amount of domain specific knowledge that should be represented in the AST is certainly not necessarily the same, one of the goals of this work is to explore this boundary and the choice of what kind of information is beneficial for calculating a patch between these two structures.

In the process of transforming one tree into another we need to keep track of three different things: changes on the constructor level (these are

the internal nodes of the trees), changes on the product level (which describe changes on the branches that go from a node to its sons), and finally changes on each element of the product, the atoms, which are either recursive changes on other nodes, or changes on leaves.

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

5.1 Spine

Calculating a spine for two trees loosely corresponds to calculating the longest common prefix between two strings. Recall that the two trees x and y are viewed as SoP, in this sense calculating the spine between x and y corresponds to capturing the common coproduct structure between them. If we think of x and y as two nodes of a tree we will have three cases to consider.

- x = y
- x and y have the same constructor but not all the subtrees are equal
- 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 trivially a copy, if the top level constructors match, the spine consists of this information and a way to transform the pairs of constructor fields and lastly if two constructors don't match, the spine must record this and also contain a way to transform the list of source fields into the list of destination fields.

The third parameter of the spine (the `U`) represents the underlying type for which we are trying to compute the transformation, the other two parameters, `al` and `at` are respectively: a function between products that describes what to do with the different constructor fields and a predicate between atoms which describes what to do with the paired fields in case we have

the same constructor. 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. The `Scns` constructor corresponds to the second case: the first argument records the common constructor for the two elements and the second represents the list of paired atoms to which we apply the `at` predicate. The `Schg` constructor represents a change of constructor: the first two arguments record the source and destination constructors, and the third argument is the `al` function applied to the constructor fields of the source and destination constructor respectively.

The spine tracks only relationships between the sum types but information about changes on the product level must be carried along too. In the case that the constructor has remained the same, we can group up the pairs of arguments and proceed from there; however, in the case the external constructor has changed, there is no obvious way of pairing up the arguments (they might be completely in different numbers and types), this motivates the following definition of alignments.

5.2 Alignment

As mentioned in the previous paragraph, the spine takes care of matching the constructors of two trees, beyond that we still need to define a way to proceed with the diff between the products of data stored in 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 describe how to change the source list into the destination. In computing 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.

This approach is inspired by the way an edit script between two strings can be computed, there is one major difference though: while in the string case we can assume deletions and insertions to be somewhat equivalent in cost (thus we can safely try to maximise one of the two) in our case, where the elements we are inserting or deleting are subtrees of arbitrary size it is not obvious if we should try and maximise insertions or deletions.

The solution to this problem is simple at this step: we simply enumerate all possible alignments, avoiding to skew the algorithm into preferring insertions over deletions or vice versa.

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

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

`A0` represents the empty alignment, `Ains` and `Adel` take as first argument a singleton representing a runtime instance of the type u and, 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, d and the other, as for the case of insertions and deletions, is simply an alignment for the tail of the list.

5.3 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. Here 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 is simply 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 can 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 UsingI u -> At recP u
```

5.4 Recursive alignments

Finally we have to define a datatype to represent changes over our recursive elements. We mirror the treatment of alignment for list of atoms except, on this level, we match, insert or delete constructors instead of atoms. 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 are inspired by zippers: 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 Almu :: U -> U -> * where
  Alspn :: Spine (At Almu) (Al (At Almu)) u -> Almu u u
  Alins :: ConstrFor v s -> Ctx (Almu u) (TypeOf s) -> Almu u v
  Aldel :: ConstrFor u s -> Ctx (Almu v) (TypeOf s) -> Almu u v
```

5.5 Putting everything together

With these types it is easy to write a function that computes the diff between two trees. We can start writing this function from the "bottom up" with the definition of a diff between two atoms. The function 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 wether 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]
```

Which 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 optimisation 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.

Optimisation Given the `Al` and `Almu` type defined in the previous paragraphs we are still left with the problem to computing these two level 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; 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 subtrees we are considering.

There is one optimisation we can introduce, despite this limitation: in the case where we can match a pair of elements then we can avoid computing an insertion followed by a deletion (resp. a deletion followed by an insertion) since the case in which we match is at least "as good" as the case in which we perform the two different operations in sequence, regardless of the actual cost we are assigning to each operation.

To implement this we must add a parameter that tracks the operation that was taken at the previous step, we will call this the `Phase`. We can define an `alignOpt` function with the same signature as `align` but parametrised with the `Phase`. The optimised version will simply avoid performing an insertion if the last step was a deletion and vice versa. After every successful match, we will call `alignOpt` to enumerate the alternatives for both insertions and deletions. In the case in which we could not match, we don't want to attempt both a deletion followed by an insertion and insertion followed by a deletion, as the ordering between the two does not really matter. To resolve this we will simply decide that we will only try deletions followed by insertions, and not the other way around.

The optimisation con 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.

6 Evaluation process

Having developed a general framework to compute patches between typed trees the goal is to 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; it is also clear that the parser should capture as much

syntactical information as possible, in order to produce code that strives to respect any syntactical convention embraced by the authors.

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.

For each file where a conflict may arise we want to find a common ancestor between the branches and calculate two sets of patches, transforming this file into the two versions that are currently present at the top of their respective tree. This operation will yield three files, the common ancestor (O) and the two versions from the conflicting branches (A and B). Before feeding this files to the algorithm we take one additional step by running diff3 on the three files and picking out just the fragments of these files that generated the merge conflicts, we then expand these fragments to the full subexpression that contains them ending up with three new files (O1, A1, B1) which contain only the Clojure expressions that turned out to be conflicting during the merge process.

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. Unfortunately tests on real world data are still out of reach in the current implementation, the sheer number of patches to enumerate is still to big, even with the optimisation described above, and the vast majority of tests ran on real world data simply consume too much memory and get terminated by the OS.

While this data set turned out not to be useful in the current iteration it still provides a good benchmark to keep testing different ideas that can make the problem more tractable. The next section will describe possible future work that can be done on the algorithm but the main line of improvement will be to optimise the algorithm enough to make it handle this data set.

7 Future Work

The optimisation described in the previous section is probably still not suited for large real world applications. Even with the optimisation the framework can only handle relatively small non-pathological inputs (in the order of 10-15 lines), clearly this is still not good enough to perform experiments on real world data. One possible direction in which to focus future work 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 `diff3` 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 `diff3` and may be able to speed up the computation to handle larger inputs.

We aim to define a notion of disjointness between patches which encodes the fact that the two patches don't apply conflicting transformations between each other. This property is key in ensuring that two patches can be safely merged. as e expect disjoint patches to commute in the order they can be applied. Experimentation may yield a counterexample to this conjecture but on the other hand could also provide good insight into the exact notion of disjointness that is needed between patches.

The resulting patch objects that are generated are very complex and contain deeply nested information which is not easy to pick up at a glance. Existing diff tools often prepend a plus or minus sign at the beginning of a line to signal it being deleted or inserted and have custom notation for conflicts. In our case the problem is complicated by the fact that information can not only be displayed along the different lines but also inside of them. It would be useful to have a representation of these patches which conveys the information they represent in an easy to digest way, this representation could as well be interactive (possibly in HTML) in order to facilitate the navigation through the different levels of the patch.

Finally we want to explore with different heuristics to score patches, this will have both the advantage of allowing us to greedily prune the search space for alignments (both recursive and non) and possibly also correlate with disjointness, in the sense that on optimal patch should strive to be minimal, and, as such, as disjoint as possible from every other patch with the same source.

Ideally all these points should be completed in future work, one consideration that can be made is that both the oracle approach and the cost heuristic should be considered of primary importance, as the results from

experimenting on concrete data sets (or lack of results) clearly point out. A nice visual representation of patches is independent of the rest but would probably help all future work on the subject, since it makes debugging and experimenting easier to mentally parse and follow. Finally, exploring the definition of disjointness, however central to the original problem of merges, is something that makes more sense once the algorithm is running in acceptable time, and as such should be postponed in favour of exploring the optimisations.

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 Lh. "The semantics of version control." Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming Software. ACM, 2014.

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

[7] ???

[8] de Vries, Edsko, and Andres Lh. "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.