# Brief Article

The Author

## 1 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 extension which duplicates an ordinary data type, such as

```haskell
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 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.

```haskell
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 concatenating an element of type `x` to 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.

```haskell
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, infact the types `K` 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`. {insert Fig 1 from [?]}

We can think of the DataKinds extension as a way to embedding dynamic information into the static fragment of the language. Singletons 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: it has 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, [?] shows us how to emulate implicit types via type classes and essentially shows how all kinds of quantification, modulo some boilerplate, are possible in Haskell.

## 2 Sum of Products

Another key idea that we will explore is related to the representation of generic datatypes. The basic idea is to view any term as the result of applying one constructor of the type the term inhabits to a list of arguments (which may also be other terms of the same type). It is useful to think about the choice of constructor and the choice of arguments as 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 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 can be easily seen to be isomorphic to the original type, we can eventually reconstruct the desired term once we are done acting on the representation.

# 3 Type-directed diff

The approach in [**?**] 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 of [**?**] 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.

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.

We will briefly sketch the process outlined in [**?**], it will be covered in more detail in the following section about the implementation. In the process of transforming one tree into another we need to keep track of three different things: changes on the constructor level, changes on the product (the arguments to the constructors) level, and finally changes on each element of the product (we will call these atoms).

### 3.1 Spine

Calculating a spine for two trees 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. In general, we 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 three different constructors for the Spine datatype, each corresponding to one of the cases described above.

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.

### 3.2 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 continue diffing 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 these two. 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 maximize one of the two) in our case, where the elements we are inserting or deleting are arbitrary big subtrees it is not obvious if we should try and maximize 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 viceversa.

### 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. 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 possibly recursive datatype we are essentially left with the problem we started from: transforming a value of a datatype into another. To do so, we simply start all over again, recursively computing a spine and an alignment between constructor fields.

## 4 Implementation

The AST defined in the Parser module represents the parse result of clojure source code. To set up the stage for the rest of the algorithm we must start building up our universe from the AST, this process is completely mechanical and could be automated via template haskell.

### 4.1 Setting up the universe

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. This is a simple example of such a family

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

data SExprList = SNil | SCons SExpr SExprList
  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 | KSExprList
```

We will also need to associate with each atom a singleton, this will allow us to relate the atoms back to the constructors in the original family.

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

Now, we can group all the constructors appearing in our original family of datatypes under the `Constr` type, in a similar way that we did for the atoms. To makes things easier to follow we prepend to the name of each constructor a tag representing which family the constructor came from.

In this case `C1` is `SExpr` and `C2` is `SExprList`.

```
data Constr =
    C1List
  | C1Operation
  | C1Value
  | C2SNil
  | C2SCons
```

Now we just need a way to relate constructors to the correct datatype in the family. To achieve this, we define the `ConstrFor` datatype, which can be viewed as a proof that a certain constructor builds element of a certain family.

```
data ConstrFor :: U -> Constr -> * where
  C1ListProof       :: ConstrFor KSExpr C1List
  C1OperationProof :: ConstrFor KSExpr C1Operation
```

```
C1ValueProof    :: ConstrFor KSExpr C1Value
C2SNilProof     :: ConstrFor KSExprList C2SNil
C2SConsProof    :: ConstrFor KSExprList C2SCons
```

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 C1List      = '[KSExprList]
  TypeOf C1Operation = '[KSExpr, KSExpr]
  TypeOf C1Value     = '[KInt]
  TypeOf C2SNil      = '[]
  TypeOf C2SCons     = '[KSExpr, KSExprList]
```

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 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
```

### 4.2 The actual puzzle

We are now going to show how to define data types to represent spines, alignments and atoms. We will also need a way to tie the recursive knot mentioned in the previous section about atoms: essentially we need to define a way to differentiate between constant and recursive atoms and treat them accordingly.

**Spine** We will start from the spine: as mentioned in the previous section, a spine represents the common structure between two elements of a type. In constructing it we must account for three different cases: 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 paris 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. This can be captured in Haskell by the following GADT:

```
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
```

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.

**Alignments** Now we need to define a type representing alignments, similarly to the spine, the alignment is parametrized by a predicate `at` which describes how to treat the underlying atoms. The specification outlined above is captured by the following GADT

```
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)
```

`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 and the other, like for insertions and deletions, is simply an alignment for the tail of the list.

**Atoms** Given to atoms x and y we have two cases: they can either be both recursive elements of the language: in which case we have to proceed recursively computing a diff between them, or, they might both be constant values: in that case we can simply build a pair out of them relating the source and destination value.

To represent these pairs 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 in the previous section, we will have the following class and instances defined

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

**Recursive alignments** Finally we have to define a datatype to represent changes over our recursive elements. We mirror the treatment of alignement for list of atoms but instead, 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 complex type with a hole somewhere; the hole will be 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
```

```
AIdel :: ConstrFor u s -> Ctx (Almu v) (TypeOf s) -> Almu u v
```

With these types it is easy to write a function that computes the diff between two trees: the lingering problem however is due to the great combinatorial explosion of possibilities that arises in computing the alignments for constructors and products. The following paragraph will describe an optimization we can employ to prune the number of possible alignments. This optimization will 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 optimizations that may be explored in future work.

**Optimization** 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 apriori 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 optimizing towards insertions or deletions. It is easy to see that in some cases, prioritizing 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 optimization 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 parametrized with the `Phase`. The optimized version will simply avoid performing an insertion if the last step was a deletion and vice versa. After every succesfull match, we will call `alignOpt` to enumerate the alternatives for both insertions and deletions. In the case in which we could not match, we still don't want to attempt both a deletion-insertion and insertion-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 optimization 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.

# 5 Future Work

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, this parser can be somewhat different to a parser 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 optimizations; 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.

To look at the extremes of this spectrum: a parser that ignores all domain specific knowledge and just parses line of text, would reduce the problem to the standard diff3 approach, not giving us more information to exploit.

## 5.1 Evaluation process

In order to test the framework in a real world context we need to find some suitable data. To aquire this we explored all the Clojure repositories on Github and extraced 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 a way to hopefully select code 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 conflit 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.

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. We expect disjoint patches to commute in the order they can be applied, experimentation could 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.

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.

# 6 Notes

Evaluation process:

- find suitables clojure repos to explore
- find all merges of two branches in git log
- for each merge, whenever we have a file appearing in both branches (file A B)
- Find comon parent of two branches where the file is present, that version of the file will be O
- p1 = All Diff between O and A
- p2 = All Diff between O and B
- Check if they are disjoint

  Check if score produces patches that are more likely to be disjoint
- If they are disjoint check if apply p1 (apply p2 O) = apply p2 (apply p1 O)

join clojure + rest phrase merging in terms of disjointness of patches calculate patches between origin and version, if disjoint check we can apply in both direction statistics about how many are actually mergable