Summary - Related Work

Jort van Gorkum

January 12, 2022

Chapter 1

An Efficient Algorithm for Type-Safe Structural Diffing

1.1 Abstract

Effectively computing the difference between two versions of a source file has become an indispensable part of software development. The de facto standard tool used by most version control systems is the UNIX diff utility, that compares two files on a line-by-line basis without any regard for the structure of the data stored in these files. This paper presents an alternative datatype generic algorithm for computing the difference between two values of any algebraic datatype. This algorithm maximizes sharing between the source and target trees, while still running in linear time. Finally, this paper demonstrates that by instantiating this algorithm to the Lua abstract syntax tree and mining the commit history of repositories found on GitHub, the resulting patches can often be merged automatically, even when existing technology has failed.

1.2 Introduction

- A consequence of the by line granularity of the UNIX diff is it inability to identify more fine-grained changes in the objects it compares.
- Ideally, however, the objects under comparison should dictate the granularity of change to be considered. This is precisely the goal of structural differencing tools.
- In this paper we present an efficient datatype-generic algorithm to compute the difference between two elements of any mutually recursive family
- The *diff* function computes these differences between two values of type a,
- and *apply* attempts to transform one value according to the information stored in the *Patch* provided to it.
- We expect certain properties of our diff and apply functions.
  - The first being *correctness*: the patch that $diff\ x\ y$ computes can be used to faithfully reproduces y from x.

$$\forall\, x\ y\ .\ apply(diff\ x\ y)\ x\ \equiv\ Just\ y$$

  - The second being *preciseness*:

$$\forall\, x\ y\ .\ apply(diff\ x\ x)\ y\ \equiv\ Just\ y$$

  - The last being *computationally efficient*: Both the *diff* and *apply* functions needs to be space and time efficient.

- There have been several attempts at generalizing UNIX diff results to handle arbitrary datatypes, but following the same recipe: enumerate all combinations of insertions, deletions and copies that transform the source into the destination and choose the 'best' one. We argue that this design has two weaknesses when generalized to work over arbitrary types:
  - The non-deterministic nature of the design makes the algorithms inefficient.
  - There exists no canonical 'best' patch and the choice is arbitrary.

- This paper explores a novel direction for differencing algorithms: rather than restricting ourselves to *insertions, deletions,* and *copy operations*, we allow the *arbitrary reordering, duplication,* and *contraction of subtrees.*

1.3 Tree Diffing: A Concrete Example

- We explicitly model permutations, duplications and contractions of subtrees within our notion of *change*. Where contraction here denotes the partial inverse of a duplication.

- The representation of a *change* between two values of type `Tree23`, is given by identifying the bits and pieces that must be copied from source to destination making use of permutations and duplications where necessary.

- A new datatype `Tree23C` φ, enables us to annotate a value of `Tree23` with holes of φ. Therefore, `Tree23C MetaVar` represents the type of `Tree23` with holes carrying metavariables.

- These metavariables correspond to arbitrary trees that are *common subtrees* of both the source and destination of change.

- We refer to a value of `Tree23C` as a *context*.

- A *change* in this setting is a pair of such contexts. The first context defines a pattern that binds some metavariables, called the **deletion context**; the second, called the **insertion context**, corresponds to the tree annotated with the metavariables that are supposed to be instantiated by the binding given by the deletion context.

```
type Change23 φ = (Tree23C φ, Tree23C φ)
```

- Applying a change is done by instantiating the metavariables in the deletion context and the insertion context:

```
applyChange :: Change23 MetaVar -> Tree23 -> Maybe Tree23
applyChange (d, i) x = del x >>= ins i
```

- The changeTree23 function merely has to compute the deletion and insertion contexts

```
changeTree23 :: Tree23 -> Tree23 -> Change23 MetaVar
changeTree23 s d = (extract (wcs s d) s, extract (wcs s d) d)
```

- The extract function receives an oracle and a tree. It traverses its argument tree, looking for opportunities to copy subtrees.

```
extract :: (Tree23 -> Maybe MetaVar) -> Tree23 -> Tree23C MetaVar
extract o t = maybe (peel t) Hole (o t)
    where peel Leaf = LeafC
          peel (Node2 a b)
            = Node2C (extract o a) (extract o b)
          peel (Node3 a b c)
            = Node3C (extract o a) (extract o b) (extract o c)
```

- This iteration of the changeTree23 function has a subtle bug: not all common subtrees can be copied. In particular, we cannot copy a tree t that occurs as a subtree of the source and destination, but also appears as a subtree of another, larger common subtree.
- One way to solve this is to introduce an additional post-processing step that substitutes the variables that occur exclusively in the deletion or insertion context by their corresponding tree.

```
changeTree23 :: Tree23 -> Tree23 -> Change23 MetaVar
changeTree23 s d
    = postprocess s d (extract (wcs s d) s) (extract (wcs s d) d)
```

1.3.1 Minimizing Changes

- The process of minimizing and isolating the changes starts by identifying the redundant part of the contexts. That is, the constructors that show up as a prefix in both the deletion and the insertion context.
- They are essentially being copied over, and we want to make this fact explicit by separating them into what we call the *spine* of the patch.
- If a constructor is in the spine, we know it has been copied, if it shows up in a change, we know it was either deleted or inserted.

```
type Patch23 = Tree23C (Change23 MetaVar)

patch :: Patch23
patch = Node3C (Hole (Hole 0, Hole 0))
               (Hole (Node2C (Hole 0) (Hole 1)
                     , Node2C (Hole 1) (Hole 0)))
               (Node2C (Hole (tree23ToC w, tree23ToC w'))
                       (Hole (Hole 3, Hole 3)))
```

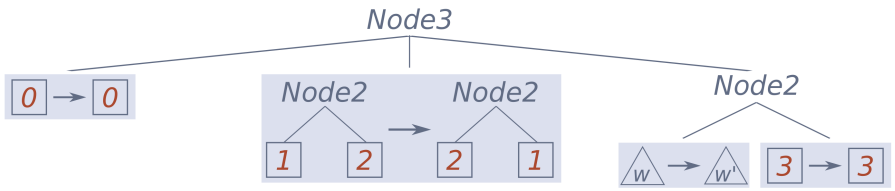

Figure 1.1: Patch23 Example

- A patch consists in a spine with changes inside it. Figure 1.1 illustrates a value of type Patch23, where the changes are visualized with a shaded background in the leaves of the spine.
- The first step to compute a patch from a change is identifying its spine.

- We are essentially splitting a monolithic change into the greatest common prefix of the insertion and deletion contexts, leaving smaller changes on the leaves of this prefix:

```
gcp :: Tree23C var -> Tree23C var -> Tree23C (Change23 var)
gcp LeafC LeafC = LeafC
gcp (Node2C a1 b1) (Node2C a2 b2)
    = Node2C (gcp a1 a2) (gcp b1 b2)
gcp (Node3C a1 b1 c1) (Node3C a2 b2 c2)
    = Node3C (gcp a1 a2) (gcp b1 b2) (gcp c1 c2)
```

- The greatest common prefix consumes all the possible constructors leading to disagreeing parts of the contexts where this might be too greedy.
- To address this problem, we go over the result from our call to `gcp`, pulling changes up the tree until each change is closed, that is, the set of variables in both contexts is identical. We call this process the closure of a patch
- The final diff function for `Tree23` is then defined as follows:

```
diffTree23 :: Tree23 -> Tree23 -> Patch23
diffTree23 s d = closure $ gcp $ changeTree23 s d
```

1.3.2 Defining the `wcs` for `Tree23`

- In order to have a working version of our diff algorithm for Tree23 we must provide the wcs implementation. Recall that the wcs function, *which common subtree*, has type:

```
wcs :: Tree23 -> Tree23 -> Tree23 -> Maybe MetaVar
```

- Given a fixed s and d, *wcs* s d x returns Just i if x is the i^{th} subtree of s and d and Nothing if x does not appear in s or d.
- To tackle the first issue and efficiently compare trees for equality we will be using cryptographic hash functions to construct a fixed length bitstring that uniquely identifies a tree modulo hash collisions.
- Said identifier will be the hash of the root of the tree, which will depend on the hash of every subtree, much like a Merkle tree

```
merkleRoot :: Tree23 -> Digest
merkleRoot Leaf = emptyDigest
merkleRoot (Node2 x y)
    = hash (concat ["node2", merkleRoot x, merkleRoot y])
merkleRoot (Node3 x y z)
    = hash (concat ["node3", merkleRoot x, merkleRoot y, merkleRoot z])
```

- the ($\equiv$) definition above is still linear, we recompute the hash on every comparison. We fix this by caching the hash associated with every node of a Tree23. This is done by the decorate function

```
data Tree23H = LeafH
             | Node2H (Tree23H, Digest) (Tree23H, Digest)
             | Node3H (Tree23H, Digest)
                      (Tree23H, Digest)
                      (Tree23H, Digest)
```

- This enables us to define a constant time merkleRoot function, shown below, which makes the ($\equiv$) function run in constant time.

```
merkleRoot :: Tree23H -> Digest
merkleRoot LeafH = emptyDigest
merkleRoot (Node2H (_, hx) (_, hy))
    = hash (encode "2" ++ hx ++ hy)
merkleRoot (Node3H (_, hx) (_, hy) (_, hz))
    = hash (encode "3" ++ hx ++ hy ++ hz)
```

- The second source of inefficiency is enumerating all possible subtrees, which can be addressed by choosing a better data structure.
- Given that a Digest is just a [Word], the optimal choice for such "database" is a Trie, mapping a [Word] to a MetaVar. Trie lookups are efficient and hardly depend on the number of elements in the trie.

1.4 Tree Diffing Generically

- Take the Tree23 type, its structure can be seen in a sum-of-products fashion through the Tree23SOP type given below.

```
type Tree23SOP = `[`[]                 -- Leaf
                 , `[I 0, I 0]         -- Node2
                 , `[I 0, I 0, I 0]] -- Node3
```

- The outer list represents the choice of constructor, and packages the sum part of the datatype whereas the inner list represents the product of the fields of a given constructor.
- The ` notation represents a value that has been promoted to the type level
- The atoms, in this case only `I 0`, represent a recursive position referencing the first type in the family.
- The `generics-mrsop` uses the type `Atom` to distinguish whether a field is a recursive position referencing the n-th type in the family, I n, or an opaque type, for example, `Int` or `Bool`, which are represented by `K KInt`, `K KBool`.
- Let us now take a mutually recursive family with more than one element and see how it is represented within the generics-mrsop framework.

```
data Zig = Zig Int  | ZigZag Zag
data Zag = Zag Bool | ZigZag Zig

type ZigCodes = `[`[`[K KInt],  `[I 1]]
                ,`[`[K KBool], `[I 0]]]
```

- The code acts as a recipe that the representation functor must follow in order to interpret those into a type of kind *.
- The representation is defined by the means of n-ary sums (NS) and products (NP) that work by induction on the codes and one interpretation for atoms (NA).
- We define the representation functor Rep as the composition of the interpretations of the different pieces:

```
type Rep φ = NS (NP (NA φ))
```

- Finally, we tie the recursive knot with a functor of kind `Nat -> *` that is passed as a parameter to *NA* in order to interpret the recursive positions.

```
newtype Fix (codes :: `[`[`[Atom]]]) (ix :: Nat)
    = Fix { unFix :: Rep (Fix codes) (Lkup codes ix) }
```

- Here, `Lkup codes ix` denotes the type level lookup of the element with index `ix` within the list codes.
- We start defining the generic notion of context, called `Tx`. Analogously to `Tree23C`.
- `Tx` enables us to augment mutually recursive family with type holes.
- We can read `Tx codes φ at` as the element of the mutually recursive family `codes` indexed by `at` augmented with holes of type φ. Its definition follows:

```
data Tx :: [[[Atom]]] -> (Atom -> *) -> Atom -> * where
    TxHole :: φ at -> Tx codes φ at
    TxOpq  :: Opq k -> Tx codes φ (K k)
    TxPeel :: Constr (Lkup codes i) c
           -> NP (Tx codes φ) (Lkup (Lkup codes i) c)
           -> Tx codes φ (I i)
```

- Looking at the definition of `Tx`, we see that its values consist in either a typed hole, some opaque value, or a constructor and a product of fields.
- The `Tx` type is, in fact, the indexed free monad over the `Rep`.
- Essentially, a value of type `Tx codes φ at` is a value of type `NA (Fix codes) at` augmented with holes of type φ.

1.4.1 Generic Representation of Changes

- With a generic notion of contexts, we can go ahead and define our generic Change type.

```
data Change codes φ at = Change (Tx codes φ at) (Tx codes φ at)
```

- The interpretation for the metavariables, `MetaVar`, now carries the integer representing the metavariable itself but also carries information to identify whether this metavariable is supposed to be instantiated by a recursive member of the family or an opaque type.

```
data MetaVar at = MetaVar Int (NA (Const Unit) at)
```

- The type of changes over `Tree23` can now be written using the generic representation for changes and metavariables.

```
type ChangeTree23 = Change Tree23Code MetaVar (I 0)
```

- We can read the type above as the type of changes over the zero-th `(I 0)` type within the mutually recursive family `Tree23Code` with values of type `MetaVar` in its holes.

1.5 Defining the Generic Oracle

- The *synthesize* function is just like a catamorphism, but we decorate the tree with the intermediate results at each node, rather than only using them to compute the final outcome. This enables us to decorate each node of a `Fix codes` with a unique identifier by running the generic decorate function, defined below.

```
newtype AnnFix x codes i
    = AnnFix (x i, Rep (AnnFix x codes) (Lkup codes i))

decorate :: Fix codes i -> AnnFix (Const Digest) codes i
decorate = synthesize authAlgebra

merkleRoot :: AnnFix (Const Digest) codes i -> Digest
merkleRoot (AnnFix (Const r, _)) = r
```

- Here, `AnnFix` is the cofree comonad, used to add a label to each recursive branch of our generic trees.
- The `Exist` datatype simply encapsulate the `ix` type index from a `Fix codes ix` into an existential one.

1.6 Merging Patches

- The merging problem is the problem of computing a new patch, $q \,//\, p$, given two patches p and q. It consists in a patch that contains the changes of q adapted to work on a value that has already been modified by p.
- If p and q are disjoint, then $p \,//\, q$ can return p without further adaptations. Our algorithm shall merge only disjoint patches, marking all other situations as a conflict.
- Our merging operator, (//), receives two patches and returns a patch possibly annotated with conflicts. We do so by matching the spines, and carefully inspecting any changes where the spines differ.

```
p // q = txMap (uncurry' reconcile) $ txGCP p q
```

- Here, the `reconcile` function shall check whether the disagreeing parts are disjoint

1.7 Experiments

- Merge Evaluation. We were able to solve a total of 66 conflicts automatically, amounting to 11% of all the conflicts we encountered.
- Performance Evaluation. In order to evaluate the performance of our implementation we have timed the computation of the two diffs, `diff o a` and diff o b, for each merge conflict `a, o, b` in our dataset.
- The results were expected given that we have seen how diff x y runs in O(n + m) where n and m are the number of constructors in x and y abstract syntax trees, respectively.
- We see that around 90% of our dataset falls within a one-second runtime.

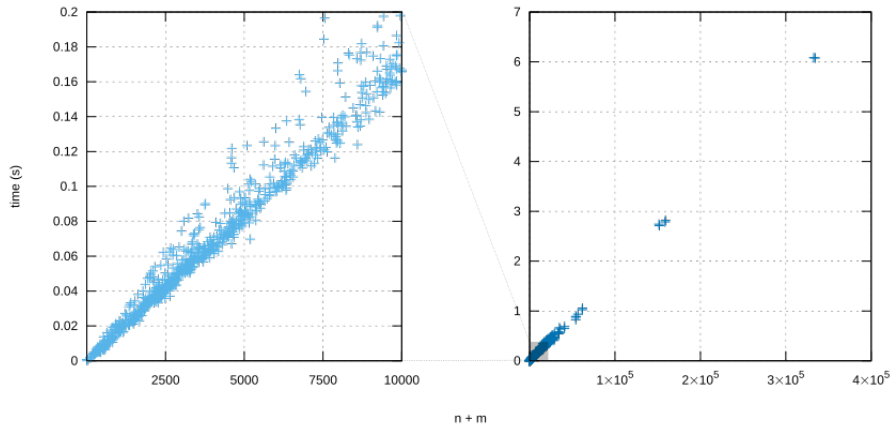

Figure 1.2: Plot of the time for diffing files

- There are two main threats to the validity of our empirical results.
  - Firstly, we are diffing and merging abstract syntax trees, hence ignoring comments and formatting.
  - Secondly, a significant number of developers prefer to rebase their branches instead of merging them. Therefore, we may have missed a number of important merge conflicts that are no longer recorded, as rebasing erases history.

1.8 Future Work

- One interesting direction for further work is how to control the sharing of subtrees. As it stands, the differencing algorithm will share every subtree that occurs in both the source and destination files. This can lead to undesirable behavior.
- Better Merge Algorithm.
- Extending the Generic Universe.

1.9 Conclusion

Throughout this paper we have developed an efficient type-directed algorithm for computing structured differences for a large class of algebraic datatypes, namely, mutually recursive families. This class of types can represent the abstract syntax tree of most programming languages and, hence, our algorithm can be readily instantiated to compute the difference between programs written in these languages. We have validated our implementation by computing diffs between Lua source files obtained from various repositories on GitHub; the algorithm's run-time is competitive, and even a naive merging algorithm already offers a substantial improvement over existing technology, for the former is tree-based and the latter is line-based. Together, these results demonstrate both a promising direction for further research and a novel application of the generic programming technology that is readily available in today's functional languages.

Chapter 2

Sums of Products for Mutually Recursive Datatypes

2.1 Abstract

Generic programming for mutually recursive families of datatypes is hard. On the other hand, most interesting abstract syntax trees are described by a mutually recursive family of datatypes. We could give up on using that mutually recursive structure, but then we lose the ability to use those generic operations which take advantage of that same structure. We present a new approach to generic programming that uses modern Haskell features to handle mutually recursive families with explicit sum-of-products structure. This additional structure allows us to remove much of the complexity previously associated with generic programming over these types.

2.2 Introduction

- The novelty in our work is in the intersection of both the expressivity of multirec, allowing the encoding of mutually recursive families, with the convenience of the more modern generics-sop style.
- The availability of several libraries for generic programming witnesses the fact that there are trade-offs between expressivity, ease of use, and underlying techniques in the design of such a library.

2.2.1 Explicit Recursion

- If we do not mark recursion explicitly, *shallow* encodings are our sole option, where only one layer of the value is turned into a generic form.
- The other side of the spectrum would be the *deep* representation, in which the entire value is turned into the representation that the generic library provides in one go.
  - These representations are usually more involved as they need an extra mechanism to represent recursion.
  - A *deep* encoding requires some explicit *least fixpoint* combinator - usually called *Fix* in Haskell.
- Depending on the use case, a shallow representation might be more efficient if only part of the value needs to be inspected. On the other hand, deep representations are sometimes easier to use, since the conversion is performed in one go, and afterwards one only has to work with the constructs from the generic library.

2.2.2 Sum of Products

- Most generic programming libraries build their type level descriptions out of three basic combinators
  - *constants*, which indicate a type is atomic and should not be expanded further;
  - *products* (usually written as $:*:$) which are used to build tuples;
  - and *sums* (usually written as $:+:$) which encode the choice between constructors.
- The `generic-sop` library explicitly uses a list of lists of types,
  - the outer one representing the sum
  - and each inner one thought of as products.
- The `` ` `` sign in the code below marks the list as operating at the type level, as opposed to term-level lists which exist at run-time.
  - An example of Haskell's *datatype* promotion.

```
Code_sop(Bin a) = `[`[a], `[Bin a, Bin a]]
```

- The *representation* is mapping the *codes*, of kind `` `[`[*]] `` into $*$. The *code* can be seen as the format that the *representation* must adhere to. Previously, in the pattern functor approach, the *representation* was not guaranteed to have a certain structure.

2.2.3 Mutually recursive datatypes

- Unfortunately, most of the generic programming approaches restrict themselves to *regular* types, in which recursion always goes into the *same* datatype, which is the one being defined.
- The syntax of many programming languages is expressed naturally using a mutually recursive family.
- The motivation of our work stems from the desire of having the concise structure that *codes* give to the *representations*, together with the information for recursive positions in a mutually recursive setting.

2.3 Background

2.3.1 GHC Generics

- Upon reflecting on the generic function, we see a number of issues.
  - Most notably is the amount of boilerplate to achieve a conceptually simple task. This is a direct consequence of not having access to the *sum-of-products* structure that Haskell's `data` declarations follow.
  - A second issue is that the generic representation does not carry any information about the recursive structure of the type.
  - Perhaps even more subtle, but also more worrying, is that we have no guarantees that the $Rep_{gen}\ a$ of type a will be defined using only the supported *pattern functors*.

2.3.2 True Sums of Products

- In comparison to the `GHC Generics` implementation we see two improvements.
  - We need one fewer type class, and
  - the definition is combinator-based.
- There are still downsides to this approach.
  - A notable one is the need to carry constraints around.

2.4 Explicit Fix: Diving Deep and Shallow

- We have combined the insight from the `regular` library of keeping track of recursive positions with the convenience of the `generics-sop` for enforcing a specific *normal form* on representations. By doing so, we were able to provide a *deep* encoding for free.

Chapter 3

Concise, Type-Safe, and Efficient Structural Diffing

3.1 Abstract

A structural diffing algorithm compares two pieces of tree- shaped data and computes their difference. Existing structural diffing algorithms either produce concise patches or en- sure type safety, but never both. We present a new structural diffing algorithm called *truediff* that achieves both properties by treating subtrees as mutable, yet linearly typed resources. Mutation is required to derive concise patches that only mention changed nodes, but, in contrast to prior work, true- diff guarantees all intermediate trees are well-typed. We formalize type safety, prove *truediff* has linear run time, and evaluate its performance and the conciseness of the derived patches empirically for real-world Python documents. While *truediff* ensures type safety, the size of its patches is on par with Gumtree, a popular untyped diffing implementation. Regardless, *truediff* outperforms Gumtree and a typed diffing implementation by an order of magnitude.

3.2 Introduction

- Most existing structural diffing algorithms follow an approach pioneered by Chawathe et al. Their approach represents structural patches as edit scripts, which convert a source tree into a target tree through consecutive destructive updates.
- Miraldo and Swierstra recently presented a new type-safe diffing algorithm. While this approach can capture moved subtrees, it suffers one major problem:
  - The size of the patch is proportional to the size of the input trees and must mention many unchanged nodes.
  - Consequently, any subsequent transmission or processing of the patch will essentially require a full tree traversal, even for small changes.
- Like Chawathe et al., we consider trees as mutable data and use URIs to refer to changed nodes directly. However, unlike them, we consider subtrees as linear resources and target a novel linearly typed edit script language called truechange.
- The type system of truechange ensures that:
  - each edit operation yields a well-typed tree (possibly containing holes),
  - the final tree is well-typed and has no holes, and
  - all detached subtrees are reattached or deleted.

- We use a novel strategy in *truediff* for identifying reusable subtrees that should be moved.
  - In a first step, we identify candidates as those trees that are equivalent except for literal values.
  - In a second step, we select an exact copy from the candidates if possible or otherwise adapt an imperfect candidate if needed.
- Summary:
  - We introduce a linearly typed edit script language *truechange* that treats subtrees as resources, and we prove well-typed edit scripts yield well-typed trees.
  - We develop a structural diffing algorithm *truediff* that yields concise and type-safe edit scripts and we prove it runs in linear time.
  - We implement *truediff* in Scala and provide bindings for ANTLR, treesitter and Gumtree.
  - We evaluate the conciseness and performance of *truediff* and the applicability for incremental computing.

3.3 Linearly Typed Edit Scripts by Example

- A concise representation of structural patches should only mention changed nodes, such that the patch is proportional in size to the change. Therefore, *truechange* uses URIs to refer to changed nodes directly.

3.4 *truechange*: Linearly Typed Edit Scripts

- Like previous untyped edit script languages, *truechange* edits describe destructive updates of the source tree, so that only changed nodes need to be mentioned. However, like previous type-safe edit script languages, *truechange* guarantees that each edit operation yields a well-typed tree
- A *truechange* edit script is a sequence of detach, attach, load, unload, and update operations.

3.5 *truediff*: Type-Safe Structural Diffing

- To compute the difference between a source tree this and a target tree that, *truediff* operates in four steps.
  - Prepare subtree equivalence relations
  - Find reuse candidates
  - Select reuse candidates
  - Compute edit script
- Our algorithm *truediff* generalizes this idea and uses two equivalence relations, both encoded through cryptographic hashes.
  - The first equivalence relation identifies reuse candidates.
  - The second equivalence relation identifies preferred trees among the reuse candidates.
- We found that using **structural equivalence** to identify candidates and **literal equivalence** to select preferred candidates yields very concise edit scripts.

3.6 Evaluation

- After a code change, we reparse the source file, use *truediff* to obtain an edit script, and then process the edits to trigger updates in the incrementally maintained Datalog database.

3.7 Related Work

- In designing truechange, we replaced the move operation with separate detach and attach operations. This change enabled us to formalize a type system for edit scripts that ensures all intermediate trees are well-typed.
- In the design of *truediff*, we do not rely on similarity scores but instead designate reusable trees based on structural and literal subtree equivalences.
- But hdiff has three main limitations that motivated the development of *truediff*.
  - First, hdiff assumes isomorphic subtrees are equal and can thus be shared. However, many applications consider the context surrounding a subtree (its parent, neighbors, etc.), which precludes sharing.
  - Second, the size of the patch computed by hdiff is proportional to the size of the source and target trees, despite supporting move edits.
  - And third, the running time of hdiff was unsatisfactory and precluded its application in incremental computing.
- Tree-sitter is a modern incremental parsing implementation based on the incremental LR parsing algorithm by Wagner and Graham. This algorithm tries to reuse subtrees of the previous AST, but only if their relative position has not changed.

3.8 Conclusion

We presented *truediff*, an efficient structural diffing algorithm that yields concise and type-safe patches. In comparing trees, *truediff* treats subtrees as mutable, yet linearly typed resources. As such, subtrees can only be attached once and slots in parent nodes can only be filled when they are empty. We capture these invariants in a new linearly typed edit script language truechange that we introduced and for which we proved type safety. To generate concise patches, *truediff* follows a novel strategy for identifying reusable subtrees: While other approaches rely on similarity scores, *truediff* uses efficiently computable equivalence classes to find and select reuse candidates. As our empirical evaluation demonstrates, this strategy enables *truediff* to deliver concise patches on par with the state of the art, while being an order of magnitude faster. We have adopted *truediff* to drive an incremental program analysis framework, which shows that *truediff* is useful in practice.