Concise, Type-Safe, and Efficient Structural Diffing

Sebastian Erdweg
JGU Mainz
Germany

Tamás Szabó
JGU Mainz / Workday
Germany

André Pacak
JGU Mainz
Germany

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 ensure 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, *truediff* 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.

CCS Concepts: • **Software and its engineering → Software notations and tools**.

Keywords: tree diffing, incremental computing

ACM Reference Format:
Sebastian Erdweg, Tamás Szabó, and André Pacak. 2021. Concise, Type-Safe, and Efficient Structural Diffing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453483.3454052

1 Introduction

A structural patch describes the differences between two pieces of tree-shaped data. Structural patches are useful in various contexts, including version control systems [2, 13], databases [4], software evolution analysis [7], and incremental computing [8, 9, 20]. Indeed, our driving use case is incremental program analysis, where processing a code change is usually much faster than reanalyzing the full AST. Structural diffing is a prerequisite for such incremental program analyses to discover which code changed. In this scenario, the derived patch must be concise so that only changed nodes are reanalyzed, it must be type-safe so that the patch can be applied to a typed tree representation, and it must be efficiently computable so that the diffing does not dominate incremental analysis times. Unfortunately, existing structural diffing algorithms do not meet these requirements as the following paragraphs outline.

Most existing structural diffing algorithms follow an approach pioneered by Chawathe et al. [4]. Their approach represents structural patches as edit scripts, which convert a source tree into a target tree through consecutive destructive updates. To compute an edit script, Chawathe et al. first compute a similarity score between pairs of source and target nodes. A source node is considered to match a target node if their similarity score is above some threshold. This bipartite matching forms the basis of computing the edit script: unmatched source nodes are deleted, unmatched target nodes are inserted, but matched nodes are moved. While this algorithm can yield concise patches that only mention changed nodes, it also has significant disadvantages:

- The similarity score is based on heuristics and has to be tuned to obtain satisfactory patches. Finding good similarity heuristics has sparked a whole line of research without a clear winner [4, 6, 7].
- Finding similar nodes has quadratic running time in the size of the source and target tree.
- The resulting edit script is not type safe, because it generates ill-typed intermediate trees that can only be captured by an untyped tree representation.

To illustrate the lack of type safety in this approach, consider computing the difference between the following two trees:

$$diff\left(\begin{array}{l} Add_1(Sub_2(a_3, b_4), Mul_5(c_6, d_7)), \\ Add(d, Mul(c, Sub(a, b))) \end{array}\right)$$

Here and in the remainder of this paper we annotate node URIs as subscripts on the source tree. URIs in the target tree are irrelevant, since we will reuse nodes from the source tree only. Approaches based on Chawathe et al. yield the following optimal edit script:

$$[\texttt{move}(Sub_2, Mul_5, 2), \texttt{move}(d_7, Add_1, 1)]$$

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454052

The first `move` indicates that the subtree rooted at Sub_2 should become the second child of Mul_5, whereas the other move makes d_7 the first child of Add_1. This edit script is correct because applying it to the source tree yields the target tree. It is also concise, since it only mentions changed nodes. However, the intermediate tree after the first `move` is ill-typed: $Add_1(Mul_5(c_6, Sub_2(a_3, b_4), d_7))$. Both Add_1 and Mul_5 violate their signature because they have the wrong number of children. The *Mul* node is particularly problematic because it is not clear how to represent it: its three subtrees do not fit into a binary tree representation. Indeed, these untyped edit scripts can only be executed against untyped rose trees, where a node can have any number of children.

This is a severe limitation and prevents the patching of typed tree representations, including algebraic data types. For example, an AST encoding `data Exp = ... | Mul(Exp,Exp)` is incompatible with the edit script from above: We cannot represent the intermediate ternary `Mul` node. If we cannot represent those intermediate trees, we cannot execute the edit script. Only type-safe edit scripts allow the use of typed tree representations (or encodings thereof).

Few structural diffing algorithms support the patching of typed trees so far. Lempsink et al. [12] and Vassena [22] represent type-safe patches as a list of node operations, corresponding to a pre-order tree traversal. They yield the following patch for the example from above:

$$\left[\begin{array}{l} \texttt{Cpy}, \texttt{Del}(Sub), \texttt{Del}(a), \texttt{Del}(b), \texttt{Ins}(d), \\ \texttt{Cpy}, \texttt{Cpy}, \texttt{Del}(d), \texttt{Ins}(Sub), \texttt{Ins}(a), \texttt{Ins}(b) \end{array}\right]$$

The `Cpy` operation leaves a constructor unchanged and refocuses on the subtrees, `Del` removes a node from the source tree, and `Ins` inserts a node into the source tree. Importantly, this edit script can be implemented as a type-safe tree transformation. The biggest problem with this approach is that it cannot detect moved subtrees. In our example, the diff first deletes $Sub(a, b)$ before reinserting it from scratch, yielding an unnecessarily verbose patch. Miraldo and Swierstra [13] recently presented a new type-safe diffing algorithm that represents the patch as a tree rewriting instead:

$$(Add(\#1, Mul(\#2, \#3)) \rightsquigarrow Add(\#3, Mul(\#2, \#1)))$$

The first tree pattern is matched against the source tree to bind metavariables #1, #2, and #3 to the corresponding subtrees. The second tree pattern is a template for generating the target tree based on the bound metavariables. 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.

The goal of this paper is a structural diffing algorithm that compares tree-shaped data efficiently, yields concise patches, and supports typed tree representations. Since none of the prior approaches can be adopted to achieve this, we designed a new algorithm called *truediff*. Like Chawathe et al. [4], 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*, which we also introduce in this paper. The type system of *truechange* ensures that (i) each edit operation yields a well-typed tree (possibly containing holes), (ii) the final tree is well-typed and has no holes, and (iii) all detached subtrees are reattached or deleted. This way we ensure *truediff* yields type-safe patches that are compatible with any mutable representation of typed trees.

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. As our empirical evaluation on the commit history of a popular Python repository shows, this approach yields concise edit scripts on par with Gumtree [6], a popular untyped approach in the style of Chawathe et al. [4]. However, *truediff* is type safe and runs in linear time, outperforming prior approaches by an order of magnitude. Specifically, we avoid costly similarity scores and decide tree equivalences (with and without literals) using cryptographic hashes instead.

In summary, we make the following contributions:

- 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 (Section 3).
- We develop a structural diffing algorithm *truediff* that yields concise and type-safe edit scripts and we prove it runs in linear time (Section 4).
- We implement *truediff* in Scala and provide bindings for ANTLR, treesitter, and Gumtree (Section 5).
- We evaluate the conciseness and performance of *truediff* and the applicability for incremental computing (Section 6).

Our implementation is available open source at https://gitlab.rlp.net/plmz/truediff.

2 Linearly Typed Edit Scripts by Example

The key contribution of this paper is an efficient structural diffing algorithm that generates concise and type-safe edit scripts. However, since prior edit script languages either lack conciseness or type safety, we had to design a new edit script language first. In this section, we introduce a linearly typed edit script language *truechange* by example. Consider again the diffing example from the previous section:

$$diff\left(\begin{array}{l} Add_1(Sub_2(a_3, b_4), Mul_5(c_6, d_7)), \\ Add(d, Mul(c, Sub(a, b))) \end{array}\right)$$

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. However, in contrast to prior untyped representations, *truechange* is type safe and prevents intermediate ill-typed trees. The minimal *truechange* edit script looks as follows:

$$\begin{bmatrix} \texttt{detach}(Sub_2, \texttt{"e1"}, Add_1), & \texttt{detach}(d_7, \texttt{"e2"}, Mul_5), \\ \texttt{attach}(d_7, \texttt{"e1"}, Add_1), & \texttt{attach}(Sub_2, \texttt{"e2"}, Mul_5) \end{bmatrix}$$

The edit script first detaches the subtree rooted at Sub_2 from its parent Add_1, where it was attached through link `"e1"`. The link usually corresponds to the name of the parent's constructor argument. We continue to detach the subtree rooted at d_7 from Mul_5, where it was attached through link `"e2"`. Since d_7 is detached, we can now attach it somewhere else, namely to link `"e1"` of Add_1, which the first detach operation made vacant. Finally, we also reattach the subtree rooted at Sub_2, namely to link `"e2"` of Mul_5. Thus, the edit script is correct: Executing it against the source tree yields a tree identical to the target tree.

The following table illustrates the intermediate trees that occur during the execution of the above edit script. To ensure type safety, we trace two aspects: the roots of detached subtrees and the empty slots left in parent nodes.

Edit operation	Roots	Slots
Initial tree	$\{Add_1\}$	$\{\}$
$\texttt{detach}(Sub_2, \texttt{"e1"}, Add_1)$	$\{Add_1, Sub_2\}$	$\{1.\texttt{e1}\}$
$\texttt{detach}(d_7, \texttt{"e2"}, Mul_5)$	$\{Add_1, Sub_2, d_7\}$	$\{1.\texttt{e1}, 5.\texttt{e2}\}$
$\texttt{attach}(d_7, \texttt{"e1"}, Add_1)$	$\{Add_1, Sub_2\}$	$\{5.\texttt{e2}\}$
$\texttt{attach}(Sub_2, \texttt{"e2"}, Mul_5)$	$\{Add_1\}$	$\{\}$

As the table shows, `detach` adds a root and an empty slot, while `attach` removes a root and an empty slot. Indeed, the linear type system of *truechange* only allows an `attach` to an empty slot with a subtree that is currently a root. This ensures that all intermediate trees are representable: Trees may contain empty slots (represented as a `null` subtree), but a slot can be filled with at most one subtree. Note that swapping subtrees with `move` operations, which combine `detach` and `attach`, will violate this property because the first move adds a subtree to a non-empty slot. The type system of *truechange* further requires that there is a single root and no empty slots left when the edit script finishes. That is, every detached root must eventually be reattached (or deleted) and any empty slot must be filled again. This guarantees that the resulting tree is well-formed. The type system we present in Section 3 additionally guarantees that the resulting tree is well-typed by tracking the type of roots and slots.

A *truechange* edit script can not only `detach` and `attach` subtrees, but also `update` and `unload` existing nodes as well as `load` new ones. Consider the following example, which illustrates *excessive subtree demand*: b is required twice in the target tree but occurs only once in the source tree.

$$\mathit{diff}\left(\begin{matrix} Add_1(a_2, b_3), \\ Add(b, b) \end{matrix}\right) = \begin{bmatrix} \texttt{detach}(a_2, \texttt{"e1"}, Add_1), & \texttt{unload}(a_2), \\ \texttt{load}(b_4), & \texttt{attach}(b_4, \texttt{"e1"}, Add_1) \end{bmatrix}$$

The edit script detaches and unloads node a_2. Indeed, it would have been a type error to detach but not use or unload that node. The target tree demands a b in place of a_2. However, we may not reuse b_3 twice because that would violate the uniqueness of URIs. Again it would have been a type error if the edit script tried to $\texttt{attach}(b_3, \texttt{"e1"}, Add_1)$ because b_3 is not a root. Instead, the edit script continues to load a new node b and assigns it a fresh URI 4. This node is then attached to the source tree, yielding $Add_1(b_4, b_3)$

A structural diffing algorithm that targets *truechange* must generate well-typed edit scripts. In particular, the diffing algorithm must treat subtrees as resources, as enforced by *truechange*'s linear type system. Otherwise the diffing algorithm is, in principle, free to choose any sequence of edit operation that transforms the source tree into the target tree. In the following section, we formalize *truechange* and its linear type system. We turn to our key contribution in Section 4, where we present an efficient diffing algorithm *truediff* that generates well-typed *truechange* edit scripts.

3 *truechange*: Linearly Typed Edit Scripts

We introduce the linearly typed edit script language *truechange*, which is the first to combine destructive updates with type safety. 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, albeit the tree may contain holes. We describe the syntax, standard semantics, and type system of *truechange*, and we develop its metatheory.

3.1 Syntax of *truechange* Edit Scripts

A *truechange* edit script is a sequence of detach, attach, load, unload, and update operations. Each of these edit operations represent a modification of the source tree. The edit operations use URIs to identify existing nodes from the source tree as well as newly loaded nodes. The main root node of a source tree is pre-defined in *truechange*; other nodes are attached to that root node.

We present the syntax of *truechange* in Figure 1 as data types written in Scala. `EditScript` is a data type with a single constructor `EditScript` that takes a sequence of `Edit` values. `Edit` is an algebraic data type with five constructors.

- A **Detach** edit consists of a child node, a link from parent to child node, and the parent node. Applying a detach edit will disconnect the child node from its parent. Note that a `Node` comprises a constructor symbol *Tag* and a `URI`, written $Tag_{\texttt{URI}}$ in this paper.

```
class EditScript(edits: Seq[Edit])

sealed trait Edit
class Detach(n: Node, l: Link, par: Node) extends Edit
class Attach(n: Node, l: Link, par: Node) extends Edit
class Load  (n: Node, ks: Kids, ls: Lits) extends Edit
class Unload(n: Node, ks: Kids, ls: Lits) extends Edit
class Update(n: Node, old: Lits, now: Lits) extends Edit

type Node = (Tag, URI) // written Tag_URI in this paper
type Kids = Seq[(Link, URI)]
type Lits = Seq[(Link, Any)]
trait URI  // we use numbers in this paper
trait Tag  // we use symbols (no quotes) in this paper
trait Link // we use strings (with quotes) in this paper
```

Figure 1. The abstract syntax of *truechange* edit scripts.

- An **Attach** edit has the same format as a detach edit, but behaves dually: It connects the child node to the parent node via the given link.
- A **Load** edit consists of a node whose URI must be fresh, a list of the node's children, and a list of the node's literals (usually numbers and strings). The children and literals are indexed by the link that connects them to the new node. Applying a load edit makes the new node available so that it can be attached.
- An **Unload** edit has the same format as a load edit, but behaves dually: It deletes the node and marks all children as detached roots.
- An **Update** edit replaces a node's literal values with new literals, but otherwise leaves the node unchanged. In particular, the node keeps its children and is still attached to its parent.

We illustrate the syntax of *truechange* through 3 edit scripts that we apply successively, starting with the empty tree ε:

ε

$\Delta_1 =$ [Load(Var$_1$, Seq(), Seq("name"->"a")),
Load(Var$_2$, Seq(), Seq("name"->"b")),
Load(Add$_3$, Seq("e1"->1, "e2"->2), Seq()),
Attach(Add$_3$, RootLink, RootTag$_0$)]

$Add_3(Var_1$("a"), Var_2("b"))

$\Delta_2 =$ [Update(Var$_2$, Seq("name"->"b"), Seq("name"->"c"))]

$Add_3(Var_1$("a"), Var_2("c"))

$\Delta_3 =$ [Detach(Add$_3$, RootLink, RootTag$_0$),
Unload(Add$_3$, Seq("e1"->1,"e2"->2), Seq()),
Load(Mul$_4$, Seq("e1"->1,"e2"->2), Seq()),
Attach(Mul$_4$, RootLink, RootTag$_0$)]

$Mul_4(Var_1$("a"), Var_2("c"))

The first edit script Δ_1 consists of three load edits followed by a single attach to the pre-defined root node. The second edit script Δ_2 updates Var_2("b") to Var_2("c"). The third edit script Δ_3 changes $Add_3(\ldots)$ into $Mul_4(\ldots)$. Note how the unload of Add_3 marks its subtrees Var_1 and Var_2 as detached roots. Therefore, we can use Var_1 and Var_2 as subtrees when loading Mul_4.

3.2 A Standard Semantics for *truechange*

A *truechange* edit script describes modifications of structured data. As such, it does not make sense to associate one single semantics with *truechange*: Like structured data can have any number of interpretations, so can their edit scripts. However, edit scripts have an important standard semantics that we introduce here.

The standard semantics of *truechange* interprets an edit script as a patch for structured data. That is, given an edit script Δ, its standard semantics $[\![\Delta]\!]$ maps a tree to a patched tree. Let $\mathcal{T}$ be the set of all trees, then the standard semantics of an edit script is a function $[\![\Delta]\!] : \mathcal{T} \to \mathcal{T}_\perp$, which yields $\perp$ if patching fails. As we will see later, the standard semantics never fails for well-typed edit scripts. Therefore, we can define $[\![\Delta_1, \ldots, \Delta_n]\!] = [\![\Delta_n]\!] \circ \cdots \circ [\![\Delta_1]\!]$ when all Δ_i are well-typed.

The standard semantics is important for three reasons:

1. The standard semantics demonstrates that any computation $f : \mathcal{T} \to A$ over structured data can instead be defined as $f_\Delta : \Delta_1, \ldots, \Delta_n \to A$ over a sequence of edit scripts by reconstructing the structured data first. Specifically, we can define $f_\Delta(\Delta_1, \ldots, \Delta_n) = f([\![\Delta_1, \ldots, \Delta_n]\!]\ \varepsilon)$, which reconstructs the structured data from the empty tree and then applies f. This means *truechange* does not restrict the expressiveness of computations in any way.
2. We usually want to avoid reconstructing the tree and instead define $f_\Delta(\Delta_1, \ldots, \Delta_n)$ in terms of its predecessor $f_\Delta(\Delta_1, \ldots, \Delta_{n-1})$ by interpreting Δ_n directly. The standard semantics provides a correctness criterion for such incremental computations.
3. The standard semantics allows us to formalize a notion of type safety that is representative for other interpretations of edit scripts.

There are many ways to define the standard semantics. We opt for a realistic semantics that patches trees efficiently, because this reveals the most insights regarding type safety. Specifically, our semantics maintains a mutable tree together with an index of all nodes. This allows us to process edit operations in constant time.

We present our semantics as Scala code in Figure 2. Our semantics maintains a mutable tree `MTree` consisting of mutable nodes `MNode`, where links to child nodes and literals can be updated destructively. By maintaining an index from `URI` to `MNode` for all loaded nodes, we can access nodes by their

```
case class MNode(node: Node, kids: mutable.Map[Link,MNode], lits: mutable.Map[Link,Any]) // a mutable tree node
class MTree { // a mutable tree with indexed nodes for constant-time access
  val root: MNode = MNode(RootTag_null, mutable.Map(RootLink -> null), mutable.Map()) // the root node
  private val index: mutable.Map[URI, MNode] = mutable.Map((null, root)) // index of all loaded nodes

  // standard semantics: t => t.patch(Δ)
  def patch(edits: EditScript): MTree = { edits.foreach(processEdit); this }

  // applies a single edit to this tree, updating nodes and the index
  def processEdit(edit: Edit): Unit = edit match {
    case Detach(tag_node, link, ptag_parent) => index(parent).kids(link) = null
    case Attach(tag_node, link, ptag_parent) => index(parent).kids(link) = index(node)
    case Load(tag_node, kids, lits) =>
      val kidNodes = kids.map((n, uri) => (n -> index(uri))).toMutableMap
      index += (node -> MNode(tag_node, kidNodes, lits.toMutableMap))
    case Unload(tag_node, kids, lits) => index -= node
    case Update(tag_node, oldlits, newlits) => index(node).lits.updateAll(newlits)
}}
```

Figure 2. The standard semantics updates nodes destructively and maintains an index of all nodes.

URI in constant time. The root of the tree is a pre-defined node with URI `null` and a single empty slot `RootLink`.

We define the standard semantics $[\![\Delta]\!] = (\texttt{t => t.patch}(\Delta))$, that is, by applying method `patch` to the current `MTree`. But the actual modification of the tree happens in `processEdit`, which `patch` invokes for each edit. For detach, we retrieve the parent node from the index and update its `link` to `null`. However, the standard semantics does not track which nodes are detached, but relies on the type system instead. For attach, we retrieve the parent node from the index and update `link` to point to the new child. For load, we construct a new `MNode` with the given URI and tag. We look up all child URIs from the index and use them as subtrees. For unload, we simply delete the node from the index. Finally, for update, we retrieve the relevant node and update its literals.

Let us review the standard semantics by applying edit script Δ_3 from the previous subsection. The following `MTree` represents the initial tree $Add_3(Var_1(\texttt{"a"}), Var_2(\texttt{"c"}))$:

```
MNode(RootTag_null, Map(RootLink->
  MNode(Add_3, Map(
    "e1" -> MNode(Var_1, Map(), Map("name"->"a")),
    "e2" -> MNode(Var_2, Map(), Map("name"->"c"))),
    Map())),
Map())
```

At this point, the index contains entries for four keys: `null`, 1, 2, 3. The first edit of Δ_3 is a detach, for which we find the parent's URI and set its child `RootLink` to `null`. The second edit is an unload, which removes URI 3 from the index. The third edit is a load, which creates a new `MNode` and adds URI 4 to the index. The final edit is an attach, for which we find the parent's URI and set its child `RootLink` to the node of 4. Thus, we obtain the following tree, which corresponds to $Mul_4(Var_1(\texttt{"a"}), Var_2(\texttt{"c"}))$ as expected:

```
MNode(RootTag_null, Map(RootLink->
  MNode(Mul_4, Map(
    "e1" -> MNode(Var_1, Map(), Map("name"->"a")),
    "e2" -> MNode(Var_2, Map(), Map("name"->"c"))),
    Map())),
Map())
```

Our standard semantics exploits only parts of the type safety that *truechange* provides: Links are never overloaded, they point to at most one subtree at any given time. This is why we can use a simple map `Map[Link, MNode]` to identify a link's target instead of the less efficient `Map[Link, Set[MNode]]`, which edit scripts in the style of Chawathe et al. [4] would require. We define a linear type system to protect *truechange* edit scripts from such overloading in the next subsection.

3.3 A Linear Type System for *truechange*

We defined a type system for *truechange* that ensures each edit operation yields a well-typed tree. Previous type-safe edit script languages [12, 22] ensured that same property, but only track complete subtrees. In contrast to these languages, *truechange* allows edit scripts to disassemble and reassemble trees using the detach and attach edits. Therefore, the type system of *truechange* must be able to track incomplete subtrees and their types.

We solve this challenge by simultaneously tracking unattached roots and empty slots, and treating both as linearly typed resources. In particular, unattached roots and empty slots must be consumed eventually by reattaching the roots and filling the slots. Moreover, a root or slot can only be used once, which prevents the sharing of subtrees and the overloading of links. The type system of *truechange* employs standard techniques from linear type systems [23, 25] to ensure these properties.

We define the type system as a typing relation written $\boxed{\Sigma \vdash\ e : (R \bullet S) \rhd (R' \bullet S')}$, where:

- e is the edit operation whose effect we are tracking.
- Σ are the signatures of node tags, defined by
  $\Sigma\ ::= \varepsilon \mid \Sigma, tag : sig$
  $sig ::= (\langle x_1 : T_1, \ldots, x_m : T_m\rangle, \langle y_1 : B_1, \ldots, y_n : B_n\rangle) \to T$,
  where each x_i is a link to a subtree of type T_i, each y_j is a link to a literal value of base type B_j, and *tag* has type T. We write $\Sigma(tag)$ to retrieve the signature of *tag*. `RootTag` has the pre-defined signature $(\langle \texttt{RootLink} : \texttt{Any}\rangle, \langle\,\rangle) \to \texttt{Root}$.
- R are the unattached subtree roots with their type, defined as ($R ::= \varepsilon \mid R, \mathsf{uri} : T$). In the judgment, R represents the roots before edit e took place, whereas R' are the roots after executing e.
- S are the empty slots with their type, defined as ($S ::= \varepsilon \mid S, \mathsf{uri.link} : T$). In the judgment, S represents the empty slots before edit e took place, whereas S' are the empty slots after executing e.

The order of bindings in R and S is irrelevant, meaning our type system is linear but not ordered [25].

We present the typing rules for *truechange* in Figure 3. A detach of *node* from $par.x_i$ is valid if *node* is not a root yet and $par.x_i$ is not an empty slot yet. A detach then introduces these as root and empty slot, respectively, assigning them types in accordance with their signatures. Dually, an attach requires *node* to be a root and $par.x$ to be an empty slot, both of which are consumed. An attach is valid if the type of the root is a subtype of the slot's type.

A load constructs a new node and makes it available as an unattached root. However, a load is only valid if the subtrees k_i of the new node are unattached roots, so that the load can consume them. Moreover, there must be *kids* and *lits* provided for all links x_i and y_j mentioned in the tag's signature, and the *kids* and *lits* must match the node's signature. That is, the type T_i of subtree k_i must be a subtype of U_i specified by the signature, and each literal value l_j must conform to base type B_j specified by the signature. Similarly, an unload must provide *kids* and *lits* for all links x_i and y_j. However, dually to load, *node* must be an unattached root whereas the subtrees k_i may *not* be unattached roots originally. The effect of an unload is to consume *node* and to make all subtrees available as unattached roots instead. Update edits do not affect roots or slots.

Finally, we lift the typing of individual edits to full edit scripts by threading their effects. An empty edit script has no effect, whereas the effects of a non-empty edit script correspond to a sequential execution of its edits. We can thus define the well-typedness of an edit script.

Definition 3.1 (Well-typed edit script)**.** An edit script Δ is well-typed if it transforms a tree without leaking detached subtrees or empty slots. That is: $\Sigma \vdash\ \Delta : ((\texttt{null} : \texttt{Root}) \bullet \varepsilon) \rhd ((\texttt{null} : \texttt{Root}) \bullet \varepsilon)$. Here `null` is the URI of the pre-defined root node, which has pre-defined type `Root`.

This definition is only applicable to non-empty trees. However, initially the pre-defined root node has an empty slot `RootLink` that needs to be filled first. We provide a specialized definition for the edit script that initializes an empty tree ε:

Definition 3.2 (Well-typed initializing edit script)**.** An edit script Δ is a well-typed initializing script if it fills `RootLink` without leaking unattached subtrees or empty slots. That is:

$$\Sigma \vdash\ \Delta : ((\texttt{null} : \texttt{Root}) \bullet (\texttt{null.RootLink} : \texttt{Any})) \rhd ((\texttt{null} : \texttt{Root}) \bullet \varepsilon)$$

As the next subsection shows, throughout the execution of an edit script, all roots R represent well-typed trees. This is the distinguishing feature of *truechange* compared to untyped edit scripts.

3.4 Metatheory of *truechange*

The metatheory of *truechange* establishes a classic invariant about its standard semantics and type system: type safety. The key challenge is that the standard semantics neither tracks detached roots nor empty slots explicitly. Thus, we must establish appropriate invariants that connect the type system to the standard semantics. To this end, we make two important observations. First, while the standard semantics does not track detached roots explicitly, all detached roots occur in the node index until they are unloaded. Thus, we can reason about detached roots using the index as an indirection. Second, the standard semantics does not track empty slots, but empty slots occur as `null` pointers in the trees. We can establish this connection by defining a generalized tree typing relative to the empty slots derived by the type system.

We start by defining the generalized tree typing for `MNode` trees, which may contain empty slots.

Definition 3.3 (`MNode` typing)**.** An `MNode` n is well-typed relative to slots S, written $(\Sigma, S \vdash n : T)$, if the following three conditions hold:

1. $\Sigma(n.\texttt{tag}) = (\langle x_1 : T_1, \ldots, x_m : T_m\rangle, \langle y_1 : B_1, \ldots, y_n : B_n\rangle) \to T$,
2. for each y_j, the literal is well-typed $\vdash n.\texttt{lits}(y_j) : B_j$, and
3. for each x_i either
   a. $n.\texttt{kids}(x_i) = \texttt{null}$, $u = n.\texttt{uri}$, and $S(u.x_i) <: T_i$, or
   b. $\Sigma, S \vdash n.\texttt{kids}(x_i) : T_i'$ and $T_i' <: T_i$.

Note how we use the slots S for kids bound to `null`, but require a well-typed tree otherwise. We can lift this typing relation to `MTree`, which may contain multiple detached roots.

Definition 3.4 (`MTree` typing)**.** An `MTree` t is well-typed relative to slots S and roots R, written $(\Sigma, S, R \vdash t)$, if the following two conditions hold:

1. for all $(p.x : T_n) \in S$, $t.\texttt{index}(p)$ is defined and has link x,
2. for all $(r : T_r) \in R$, $t.\texttt{index}(r)$ is defined and well-typed $\Sigma, S \vdash t.\texttt{index}(r) : T_r$.

This typing relation establishes the required invariants about the index used in `MTree`. Now we can almost state our main theorem: Well-typed edit scripts transform well-typed `MTree`

$$\text{T-Detach}\ \frac{node \notin dom(R) \quad par.x_i \notin dom(S) \quad \Sigma(tag) = (_,_) \to T \quad \Sigma(ptag) = (\langle \ldots, x_i : T_i, \ldots \rangle, _) \to _}{\Sigma \vdash \texttt{Detach}(tag_{node}, x_i, ptag_{par}) \ : \ (R \bullet S) \rhd (R, node : T \bullet S, par.x_i : T_i)}$$

$$\text{T-Attach}\ \frac{T <: T'}{\Sigma \vdash \texttt{Attach}(tag_{node}, x, ptag_{par}) \ : \ (R, node : T \bullet S, par.x : T') \rhd (R \bullet S)}$$

$$\text{T-Load}\ \frac{\Sigma(tag) = (\langle x_1 : U_1, \ldots, x_m : U_m \rangle, \langle y_1 : B_1, \ldots, y_n : B_n \rangle) \to T \quad \forall 1 \le i \le m.\ (T_i <: U_i) \quad \forall 1 \le j \le n.\ (\vdash l_j : B_j)}{\Sigma \vdash \texttt{Load}(tag_{node}, \langle x_1 = k_1, \ldots, x_m = k_m \rangle, \langle y_1 = l_1, \ldots, y_n = l_n \rangle) \ : \ (R, k_1 : T_1, \ldots, k_m : T_m \bullet S) \rhd (R, node : T \bullet S)}$$

$$\text{T-Unload}\ \frac{\Sigma(tag) = (\langle x_1 : T_1, \ldots, x_m : T_m \rangle, \langle y_1 : B_1, \ldots, y_n : B_n \rangle) \to T' \quad \{k_1, \ldots, k_m\} \cap dom(R) = \emptyset}{\Sigma \vdash \texttt{Unload}(tag_{node}, \langle x_1 = k_1, \ldots, x_m = k_m \rangle, \langle y_1 = l_1, \ldots, y_n = l_n \rangle) \ : \ (R, node : T \bullet S) \rhd (R, k_1 : T_1, \ldots, k_m : T_m \bullet S)}$$

$$\text{T-Update}\ \frac{\Sigma(tag) = (\ldots, \langle y_1 : B_1, \ldots, y_n : B_n \rangle) \to T \quad \forall 1 \le j \le n.\ (\vdash l'_j : B_j)}{\Sigma \vdash \texttt{Update}(tag_{node}, \langle y_1 = l_1, \ldots, y_n = l_n \rangle, \langle y_1 = l'_1, \ldots, y_n = l'_n \rangle) \ : \ (R \bullet S) \rhd (R \bullet S)}$$

$$\text{T-EditScript-Nil}\ \frac{}{\Sigma \vdash \texttt{EditScript}(\texttt{Nil}) : (R \bullet S) \rhd (R \bullet S)} \qquad \text{T-EditScript-Cons}\ \frac{\Sigma \vdash e : (R_1 \bullet S_1) \rhd (R_2 \bullet S_2) \quad \Sigma \vdash \texttt{EditScript}(\Delta) : (R_2 \bullet S_2) \rhd (R_3 \bullet S_3)}{\Sigma \vdash \texttt{EditScript}(e :: \Delta) : (R_1 \bullet S_1) \rhd (R_3 \bullet S_3)}$$

Figure 3. The type system of *truechange* keeps track of unattached roots R and empty slots S.

into well-typed `MTree` under the standard semantics. As final preparatory step, we must ensure the operations of the edit script comply to the source tree syntactically: the URIs exist and have the designated tags and links.

Definition 3.5 (Syntactic compliance)**.** An edit script Δ syntactically complies with an `MTree` t, written $\Delta \prec t$, if the following conditions hold:

1. For all $\texttt{Detach}(tag_{node}, x, ptag_{par}) \in \Delta$:
   $p = t.\texttt{index}(par)$ is defined, $p.\text{tag} = ptag$, $n = p.\texttt{kids}(x)$, $n.\text{uri} = node$, and $n.\text{tag} = tag$.
2. For all $\texttt{Attach}(tag_{node}, x, ptag_{par}) \in \Delta$:
   Syntactic compliance is ensured by the type system already, no additional checks needed.
3. For all $\texttt{Load}(tag_{node}, kids, lits) \in \Delta$:
   $node$ is a fresh URI, that is, $t.\texttt{index}(node)$ is undefined and for all other $\texttt{Load}(tag'_n, _, _) \in \Delta$, $node \neq n$.
4. For all $\texttt{Unload}(tag_{node}, ks, ls)$: $n = t.\texttt{index}(node)$ is defined, $n.\text{tag} = tag$, $\forall (x_i, k_i) \in ks.\ n.\texttt{kids}(x_i).\text{uri} = k_i$, and $\forall (y_j, l_j) \in ls.\ n.\texttt{lits}(y_j) = l_j$.

With these definitions, we can now state our main theorem. Our main theorem considers the application of an edit script to a simple `MTree` t with a single root $t.\texttt{root}$ and no empty slots. We call such an `MTree` closed. Well-typed edit scripts preserve the well-typedness of closed `MTree`.

Theorem 3.6 (Type safety for closed `MTree`)**.** *Given a closed well-typed* `MTree` t *with* $\Sigma, \varepsilon \vdash t.\texttt{root} : \texttt{Root}$ *(no empty slots), and given a syntactically compliant edit script* Δ *with* $\Delta \prec t$. *If edit script* Δ *is well-typed* $\Sigma \vdash \Delta : ((\texttt{null} : \texttt{Root}) \bullet \varepsilon) \rhd ((\texttt{null} : \texttt{Root}) \bullet \varepsilon)$*, then patching succeeds* $t.\texttt{patch}(\Delta) = t'$ *and the root of* t' *is well-typed* $\Sigma, \varepsilon \vdash t'.\texttt{root} : \texttt{Root}$.

Proof. By the following Lemma 3.7 with $R = R' = (\texttt{null} : \texttt{Root})$ and $S = S' = \varepsilon$. □

We generalize this theorem to `MTree` with multiple detached roots and empty slots. We call such `MTree` open. Well-typed edit scripts preserve the well-typedness of open `MTree`.

Lemma 3.7 (Type safety for open `MTree`)**.** *Given an open well-typed* `MTree` t *with* $\Sigma, S, R \vdash t$*, and given a syntactically compliant edit script* Δ *with* $\Delta \prec t$*. If* Δ *is well-typed* $\Sigma \vdash \Delta : (R \bullet S) \rhd (R' \bullet S')$*, patching succeeds* $t.\texttt{patch}(\Delta) = t'$ *and* t' *is well-typed* $\Sigma, S', R' \vdash t'$.

Proof. Straightforward induction over Δ using the following Lemma 3.8. □

Finally, we show that each individual edit operation preserves the well-typedness of `MTree`. Thus, all intermediate trees are well-typed for *truechange*.

Lemma 3.8 (Type-safe edit)**.** *Given an open well-typed* `MTree` t *with* $\Sigma, S, R \vdash t$*, and given a syntactically compliant edit* e *with* $e \prec t$*. If edit* e *is well-typed* $\Sigma \vdash e : (R \bullet S) \rhd (R' \bullet S')$*, patching succeeds* $t.\texttt{processEdit}(e) = t'$ *and all roots* R' *are well-typed* $\Sigma, S', R' \vdash t'$.

Proof. By case distinction on e.

- e = Detach(tag_{node}, x_i, $ptag_{par}$): We have $R' = R, node : T$ and $S' = S, par.x_i : T_i$. Since $e \prec t$, t.index(par).kids(x_i) is defined and patching succeeds, yielding t' with t'.index = t.index. Let $r \in R$ be the tree that contained *node* in t. Since t.index(r) was well-typed, so is t.index(*node*) and hence t'.index(*node*), which did not change. Since $par.x_i$ was added as a slot in S', r is still well-typed in t'. In summary we get $\Sigma, S', R' \vdash t'$.
- e = Attach(tag_{node}, x, $ptag_{par}$): We have $R = R', node : T$ and $S = S', par.x : T'$. Since $\Sigma, S, R \vdash t$, t.index(par) and t.index(*node*) is defined, such that patching succeeds, yielding t' with t'.index = t.index. Since $T <: T'$, binding the slot $par.x$ to t.index(*node*) preserves well-typedness, such that $\Sigma, S', R' \vdash t'$.
- e = Load(tag_{node}, *kids*, *lits*):
  We have $R = (R_0, k_1 : T_1, \ldots, k_m : T_m)$, $R' = (R_0, node : T)$, and $S' = S$. Since $\Sigma, S, R \vdash t$, t.index(k_i) is defined and well-typed $\Sigma, S \vdash t$.index(k_i) $: T_i$ for all k_i. Therefore, the new MNode *subtree* is well-typed and patching succeeds, yielding t' with t'.index(*node*) = *subtree* and t'.index(p) = t.index(p) for $p \neq node$. Thus, we obtain $\Sigma, S', R' \vdash t'$.
- e = Unload(tag_{node}, *kids*, *lits*): We have $R = (R_0, node : T)$, $R' = (R_0, k_1 : T_1, \ldots, k_m : T_m)$, and $S' = S$. Since $\Sigma, S, R \vdash t$, t.index(*node*) is defined and patching succeeds, yielding t' with t'.index(*node*) = $\bot$ and t'.index(p) = t.index(p) for $p \neq node$. Moreover, $\Sigma, S \vdash t$.index(*node*) $: T$, such that all kids of *node* must be well-typed $\Sigma, S \vdash t$.index(k_i) $: T_i'$ and $T_i' <: T_i$. We thus obtain $\Sigma, S', R' \vdash t'$.

□

Since all intermediate trees are well-typed in *truechange* and edits never overload links, a typed representation can be chosen for trees as long as it supports empty slots. But even untyped tree representations benefit from this property because they can use a map to store link targets, as our standard semantics did. In the remainder of this paper we show an efficient structural diffing algorithm that can generate well-typed *truechange* edit scripts.

4 *truediff*: Type-Safe Structural Diffing

We present a novel structural diffing algorithm called *truediff* that generates well-typed *truechange* edit scripts. Our diffing algorithm generates concise edit scripts and efficiently runs in linear time. To compute the difference between a source tree this and a target tree that, *truediff* operates in four steps.

1. **Prepare subtree equivalence relations**: We define two equivalence relations to decide which subtrees should be reused: literal equivalence and structural equivalence, which ignores literal values. We use cryptographic hashes and hash tries to implement these relations efficiently.
2. **Find reuse candidates**: We introduce subtree shares, which manage subtrees as resources. All structurally equivalent subtrees in this and that are assigned the same share. Subtrees in this are available resources that can be reused, whereas subtrees in that are required resources.
3. **Select reuse candidates**: For every subtree in that, we try to assign an available subtree from this. If there are multiple candidates, we select a literally equivalent tree if one is available. A subtree can be assigned at most once.
4. **Compute edit script**: Finally, we compute the edit script by traversing this and that simultaneously. We only need to generate edits for changed nodes, thus we can skip all nodes for which this and that agree. For other nodes we check the subtree assignment.

In this section, we focus on the algorithmic aspects of *truediff* and consider a simple algebraic data type Exp only. Section 5 describes our datatype-generic implementation using Scala macros.

```
sealed trait Exp extends Diffable
case class Num(n: Int) extends Exp
case class Add(e1: Exp, e2: Exp) extends Exp
case class Sub(e1: Exp, e2: Exp) extends Exp
case class Call(f: String, a: Exp) extends Exp
```

We provide trait Diffable as supertype for Exp to collect all generic functionality that *truediff* requires. For now, we only define uri and tag, but will expand Diffable later:

```
trait Diffable {
  def uri: URI // the URI of this node
  def tag: Tag // the tag of this node
  ... // to be expanded later
}
```

4.1 Step 1: Prepare Subtree Equivalence Relations

A structural diffing algorithm can only generate concise patches if it detects moved subtrees. Moved subtrees can be concisely represented in *truechange* by detaching the subtree and attaching it elsewhere. But how can a diffing algorithm decide which subtrees to move?

Previous approaches in the style of Chawathe et al. [4] computed similarity scores to decide which subtrees to reuse. However, computing similarity scores has a quadratic running time, since each node of the source tree must be compared to each node of the target tree. Instead, we follow Dotzler and Philippsen [5] and Miraldo and Swierstra [13] who assign each subtree a unique cryptographic hash, such that two trees are equal if and only if their hashes are equal. Using a hash trie, we can efficiently identify all subtrees that share the same cryptographic hash.

Our algorithm *truediff* generalizes this idea and uses two equivalence relations, both encoded through cryptographic hashes. The first equivalence relation identifies reuse candidates. Any reuse candidate may be moved to match an equivalent target tree, but it might need adaptions to match it exactly. The second equivalence relation identifies preferred trees among the reuse candidates. When possible, *truediff* chooses one of the preferred candidate trees.

In principle, *truediff* is parametric with respect to these two equivalence relations. However, we found that using structural equivalence to identify candidates and literal equivalence to select preferred candidates yields very concise edit scripts. Two trees are structurally equivalent if they are equal except for literal values, that is, they have the same shape. Two trees are literally equivalent if they are equal except for node tags, that is, they have the same literals. Note that if two trees are structurally and literally equivalent, then they are equal. We extend `Diffable` to compute the corresponding cryptographic hashes for each subtree:

```
// in trait Diffable
lazy val structureHash: Array[Byte] = {
  val d = MessageDigest.getInstance("SHA-256")
  d.update(this.tag.getBytes)
  directSubtrees.foreach(t => d.update(t.structureHash))
  d.digest()
}
lazy val literalHash: Array[Byte] = {
  val d = MessageDigest.getInstance("SHA-256")
  lits.foreach(l => d.update(l.getBytes))
  directSubtrees.foreach(t => d.update(t.literalHash))
  d.digest()
}
```

For example, `Add(Num(1),Num(2))` is structurally equivalent to `Add(Num(3),Num(4))`, but not to `Sub(Num(1),Num(2))`. That is, we will consider a tree with modified literals as a reuse candidate, but not a tree with changed constructors. While `Add(Num(1),Num(2))` and `Sub(Num(1),Num(2))` have equivalent literals, we only use literal equivalence to select among the reuse candidates. That is, we prefer to reuse an exact copy of a tree that has the same structure and the same literals.

4.2 Step 2: Find Reuse Candidates

We introduce subtree shares, which we use to manage available and required subtrees during diffing. We assign a subtree share to each subtree in source tree `this` and target tree `that`. Importantly, two subtrees are assigned the same share if and only if they are structurally equivalent, which our `SubtreeRegistry` ensures. While assigning shares, we traverse `this` and `that` simultaneously to preemptively detect subtrees that can be reused without moving them at all.

```
// in trait Diffable
var share: SubtreeShare = _
def assignShares(that: Diffable, reg: SubtreeRegistry) {
  reg.assignShare(this) // sets this.share
  reg.assignShare(that) // sets that.share
  if (this.share == that.share) // true iff this≃that
    this.assignTree(that) // preemptive tree assignment
  else assignSharesRec(that, reg)
}
def assignSharesRec(that:Diffable,reg:SubtreeRegistry) {
 if (this.tag == that.tag) { // recurse simultaneously
   this.share.registerAvailableTree(this)
   val ts = this.directSubtrees.zip(that.directSubtrees)
   ts.foreach {case (l,r) => l.assignShares(r, reg)}
 } else { // recurse separately
   this.foreachTree(reg.assignShareAndRegisterAvailable)
   that.foreachSubtree(reg.assignShare)
}}
```

For example, consider `this = Add(Call("f",Num(1)),Num(2))` and `that = Add(Call("g",Num(1)),Sub(Num(2),Num(2)))`. These trees are not structurally equivalent and hence are assigned separate shares. However, both start with the `Add` tag so that we recurse into them simultaneously, considering their operands next. The two `Call` subtrees are structurally equivalent, hence we preemptively assign them and stop recursing. In contrast, the right-hand operands use different tags `Num` and `Sub`, hence we finish by recursing into them separately. Nonetheless, our `SubtreeRegistry` will assign all `Num(2)` subtrees the same share.

4.3 Step 3: Select Reuse Candidates

Step 3 finalizes which subtrees to reuse where exactly. Specifically, we compute a subtree assignment that associates subtrees of `this` to subtrees of `that` and vice versa. Importantly, we treat subtrees as linear resources that can be assigned to at most one other subtree. We represent this assignment through field `assigned` in `Diffable` and use method `assignTree` to ensure the assignment is symmetric.

```
// in trait Diffable
var assigned: Diffable = _
def assignTree(that: Diffable): Unit = {
  this.assigned = that
  that.assigned = this
}
def assignSubtrees(that:Diffable, reg:SubtreeRegistry) {
  val queue = new mutable.PriorityQueue[Diffable]()(
     Diffable.highestFirstOrdering)
  queue += that
  while (queue.nonEmpty) {
    val level = queue.head.treeheight
    val nexts = queue.dequeueWhile(_.treeheight==level)
    val unassigned = selectTrees(preferred=false, reg,
      selectTrees(preferred=true, nexts, reg))
    unassigned.foreach(queue++=_.directSubtrees)
  }
}
```

Method `assignSubtrees` traverses the subtrees of `that` to acquire subtrees of `this`. Crucially for performance, method `assignSubtrees` is greedy and never releases a tree once acquired, which is possible since we identified all reuse candidates trees in Step 2 already. We traverse the subtrees of `that` in highest-first order to avoid subtree fragmentation: We try to reuse subtrees as a whole, rather than reusing smaller fragments of it. Specifically, we dequeue all subtrees of the same height and then select reuse candidates. We first try to select preferred reuse candidates; for the remaining trees, we try to select any other reuse candidate. When no reuse

candidate can be assigned, we add the subtrees to the queue in order to find smaller subtrees that may be reusable.

We need to be careful to ensure no tree is assigned more than once. When selecting a tree to be reused, `SubtreeShare` deregister the acquired tree and its subtrees to ensure they cannot be used elsewhere. In doing so, we must also check if a subtree of the acquired tree was pre-emptively assigned in Step 2. Since we prioritize the reuse of larger subtrees, we must undo the pre-emptive assignment of smaller subtrees and mark them as required instead.

Consider again trees `this = Add(Call("f",Num(1)),Num(2))` and `that = Add(Call("g",Num(1)),Sub(Num(2),Num(2)))`. Method `assignSubtrees` traverses `that`. Since no reuse candidate is available for the `Add` node, we add its subtrees to the queue. Neither `Call` nor `Sub` have a preferred reuse candidate, but for `Call` we have a non-preferred reuse candidate in `this` that we select. Thus, we only push the subtrees of `Sub` to the queue. In the next iteration, we have two instances of `Num(2)`, but only one reuse candidate. When we select `Num(2)` from `this` the first time, `SubtreeShare` will deregister the tree from its share, so that it cannot be reused twice. Consequently, only one of the `Num(2)` from `that` will be assigned a subtree from `this`, while the other will have to be loaded afresh.

4.4 Step 4: Compute Edit Script

Finally, we compute an edit script that transforms `this` into `that`. We also produce a patched tree that uses newly loaded subtrees and subtrees from `this` only. This patched tree can be used for subsequent diffing computations. Method `computeEdits` simultaneously traverses `this` and `that` top-down as long as their tag and literals coincide. Parameters `parent` and `link` represent where we came from in `this`.

```
// in trait Diffable
def computeEdits(that: Diffable, parent: Node, link:
     Link, edits: EditBuffer): Diffable = {
 if (this.assigned!=null && assigned.uri==that.uri)
   return this.updateLits(that, edits)
 if (this.assigned == null && that.assigned == null) {
   val t = this.computeEditsRec(that,parent,link,edits)
   if (t != null) return t
 }
 // have to replace this subtree by that subtree
 edits += Detach((this.tag, this.uri), link, parent)
 this.unloadUnassigned(edits)
 val t = that.loadUnassigned(edits)
 edits += Attach((t.tag, t.uri), link, parent)
 t
}
def updateLits(that:Diffable,edits:EditBuffer): Diffable
def computeEditsRec(that: Diffable, parent: Node, link:
     Link, edits: EditBuffer) : Diffable
def loadUnassigned(edits: EditBuffer): Diffable
def unloadUnassigned(edits: EditBuffer): Unit
```

If `this` is assigned to `that` (first `if`), we can leave `this` in place and only need to update its literals (in case they are only structurally equivalent). If neither `this` nor `that` are assigned (second `if`), we try to reuse `this` node and continue the simultaneous traversal with the subtrees of `this` and `that` (method `computeEditsRec`). In all other cases, we must detach `this` and replace it by a tree identical to `that`. Recall that we treat subtrees as resources and may thus not simply discard subtree `this`. Instead, we unload it except for assigned subtrees (method `unloadUnassigned`). Then we must load a copy of `that` except for subtrees we can reuse from the source tree (method `loadUnassigned`)). Finally, we attach the copy of `that` to replace `this`. We added four abstract methods to `Diffable` that data types must implement, unless they use our macro (Section 5). We exemplify their implementation in our artifact https://gitlab.rlp.net/plmz/truediff/-/blob/pldi21-artifact/truediff/src/test/scala/truediff/manual/Exp.scala.

Consider again `this = Add`$_1$`(Call`$_2$`("f", Num`$_3$`(1)), Num`$_4$`(2))` and `that = Add(Call("g", Num(1)), Sub(Num(2), Num(2)))`. The `Add` nodes are not assigned, hence we recurse into their subtrees (second `if`). The `Call` nodes are assigned to each other (first `if`), hence we only update the literals. This yields an update of `Call`$_2$ to replace its name. We continue comparing `Num`$_4$ to the `Sub` node, which fails. Hence we detach `Num`$_4$, although we won't unload it since it is assigned. Indeed, when loading the `Sub` subtree, we will reuse `Num`$_4$ for one of its operands. Finally, we attach the loaded `Sub` node to `Add`$_1$.

A final but important remark: We emit edits by adding them into an `EditBuffer`. This `EditBuffer` not only collects edits, but distinguishes negative edits (detach and unload) from positive edits (attach and load). The final edit script will contain negative edits before positive edits. This way, we ensure a subtree is detached before it is attached, which the algorithm does not otherwise ensure.

4.5 *truediff*: Main Algorithm and Properties

We obtain the complete *truediff* algorithm by connecting all four steps.

```
// in trait Diffable
def compareTo(that: Diffable): (EditScript,Diffable) = {
  // Step 1: by construction of the Diffable data
  val subtreeReg = new SubtreeRegistry
  this.assignShares(that, subtreeReg)   // Step 2
  this.assignSubtrees(that, subtreeReg) // Step 3
  val edits = new EditBuffer
  val newtree = this.computeEdits(that, (RootTag,null),
     RootLink, edits) // Step 4
  (edits.toEditScript, newtree)
}
```

Theorem 4.1 (Linear run-time complexity). *Algorithm truediff runs in linear time. Let m be the number of nodes in* `this` *and n be the number of nodes in* `that`*. Then truediff $\in O(m+n)$.*

Proof. Each step of *truediff* runs in linear time.

- Step 1 computes two hashes for each node in `this` and `that`. Since the hash arrays have fixed size (depending on the

hashing function) and maximum branching factor is finite (and usually small), the work for each node is constant.
- Step 2 visits each node in `this` and `that` at most once to assign a share. Assigning a share takes constant time because we use a hash trie with a fixed-sized bitstring.
- Step 3 visits each node in `that` at most once to find a matching tree in `this`. To check if an available tree exists, we only need to peek at the head of `availableTrees`, which takes constant time. When no available tree can be found, we simply traverse the nodes of `that`. When an available tree `src` is found, we do not traverse into `next` any further, but instead deregister all nodes in `src`. However, since `src` is structurally equivalent to `next`, they have the same amount of nodes and their traversal incurs the same amount of work. Hence, Step 3 is linear in n.
- Step 4 visits each node in `this` and `that` at most once. Any node in `this` is either updated, compared, or unloaded, and any node in `that` is either compared or loaded.

Therefore, *truediff* runs in linear time. □

We claim the following conjectures without formal proof. We have tested them through more than 200 test cases.

Conjecture 4.2. *The edit scripts produced by truediff are type-safe. That is, if* `this.compareTo(that)=(Δ,patched)`*, then* $\Sigma \vdash \Delta : ((\texttt{null} : \texttt{Root}) \bullet \varepsilon) \rhd ((\texttt{null} : \texttt{Root}) \bullet \varepsilon)$.

Conjecture 4.3. *The edit scripts produced by truediff are correct: They transform the source tree into the target tree. Let* $t_{\texttt{this}}$ *and* $t_{\texttt{that}}$ *be the* `MTree` *corresponding to* `this` *and* `that`*. If* `this.compareTo(that)` $= (\Delta, _)$*, then* $t_{\texttt{this}}.patch(\Delta) \simeq t_{\texttt{that}}$.

5 Implementation

We implemented *truediff* in Scala as a datatype-generic algorithm. The core algorithm is implemented in trait `Diffable` as shown in the previous section. Any algebraic data type that implements `Diffable` inherits method `compareTo`, which runs *truediff*. However, `Diffable` declares a number of abstract methods that have to be implemented correctly for each constructor of the algebraic data type. To ease the application of *truediff*, we developed a Scala macro `@diffable` that implements these methods automatically [3]. We can use our macro to enable *truediff* as follows:

```
@diffable sealed trait Exp
@diffable case class Var(name: String) extends Exp
@diffable case class Add(e1: Exp, e2: Exp) extends Exp
```

Additionally, we manually implemented `Diffable` classes to wrap nodes from the popular parser frameworks ANTLR and treesitter. For example, the following code uses ANTLR to parse and *truediff* to compare two Java source files:

```
val tree1 = Java8.parseCompilationUnit(file1)
val tree2 = Java8.parseCompilationUnit(file2)
val wrap = new RuleContextMapper(Java8Parser.ruleNames)
wrap.diffable(tree1).compareTo(wrap.diffable(tree2))
```

Finally, we implemented a similar `Diffable` wrapper for nodes used by the structural diffing tool Gumtree [6]. This enables us to compare the performance and conciseness of *truediff* against Gumtree on exactly the same input trees.

6 Evaluation

We evaluate the conciseness and performance of *truediff* empirically, and demonstrate its applicability to incremental computing. We compare *truediff* to Gumtree [6] and hdiff [13]. Gumtree is a popular untyped structural diffing tool based on Chawathe et al. [4]. Hdiff is a recent typed structural diffing tool implemented in Haskell.

Setup. We benchmark real-world Python files extracted from the last 500 commits of the popular deep learning API keras,[1] starting with commit `1a3ee84`. In total, 2393 Python files were changed in these commits. We invoke *truediff*, Gumtree, and hdiff on each changed file three times and retain their fastest run. For *truediff* and Gumtree we warmed up the JIT by diffing 100 files; hdiff is optimized ahead-of-time and does not require warmup. For *truediff*, we reconstruct new trees before each invocation such that the time required for computing cryptographic hashes is taken into account. We conducted all measurements on an Intel Xeon W at 3.5 GHz with 32 GB of RAM, running 64-bit OSX 10.15.7, Java 1.11.0.5 with 8GB max heap space. The raw data is available open source at https://gitlab.rlp.net/plmz/truediff/-/tree/pldi21-artifact/benchmark/measurements.

Conciseness. We compare the difference $(a - b)$ and ratio (a/b) in patch size between *truediff*, hdiff, and Gumtree. For Gumtree and *truediff*, we count the number of edit operations. For *truediff*, we count a `Load` directly followed by an `Attach` of the same node as one edit and analogous for a `Detach` followed by an `Unload` This closely corresponds to `Ins` and `Del` edits used by Gumtree, which also un/load and de/attach at once. Note that *truediff* merges such edits into a compound edit during diffing and our running times account for the extra effort. For hdiff, we count the number of constructors mentioned in the tree rewriting.

We show the conciseness comparison as box plots in Figure 4: difference on the left, ratio on the right. *truediff* produces considerably shorter patches than hdiff, both in terms of difference and ratio. On average, hdiff patches are 18.8x larger than *truediff* patches. This empirically confirms our analysis from Section 1. But even compared to Gumtree, *truediff* fares well: On average, *truediff* patches are only 1.01x larger than Gumtree patches. Thus, *truediff* patches are on par with Gumtree regarding their size. This confirms that *truediff*'s diffing strategy based on subtree equivalences works just as well as Gumtree's similarity scores. However, note that there are outliers in both directions when comparing the absolute difference.

[1] https://keras.io/

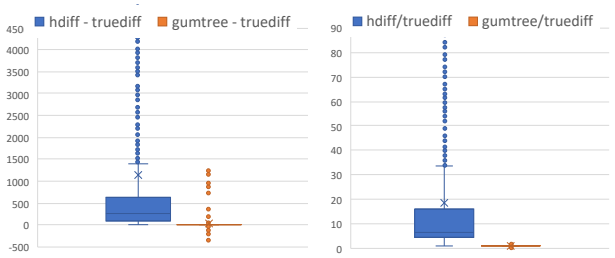

Figure 4. Edit script conciseness: patch size difference (left) and patch size ratio (right). *truediff* is on par with Gumtree, both of which yield shorter patches than hdiff.

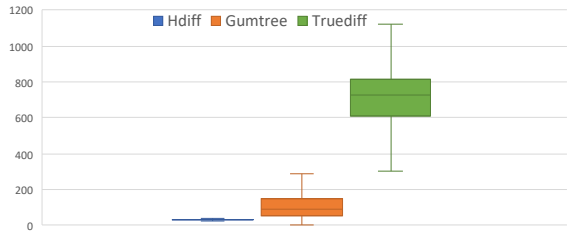

Figure 5. Diffing throughput (nodes/ms): *truediff* outperforms hdiff by ~22x and Gumtree by ~8x.

Performance. What does the type safety and conciseness of *truediff* cost in terms of performance? We try to answer this question by comparing the throughput of *truediff*, hdiff, and Gumtree. Figure 5 shows the throughput as a box plot in nodes per millisecond, excluding parsing times. Clearly, *truediff* significantly outperforms hdiff (by ~22x) and gumtree (by ~8x). The median running time of *truediff* is 6.4*ms* per file, while the mean running time is 12.7*ms*. These running times testify that *truediff* can be used for on-the-fly diffing of real-world source trees.

Incremental computing. To demonstrate that *truediff* can be used to drive incremental computing, we have reimplemented the driver of the incremental program analysis framework IncA [20]. IncA incrementally maintains a Datalog database of derived properties about a syntax tree, such as typing [15, 21] or points-to information [19]. However, IncA so far required that programs were edited in a projectional editor, which issues fine-grained change notifications. We implemented a new driver for IncA that replaces projectional editing by structural diffing to obtain AST edits. Specifically, 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. Since parsing is fast, *truediff* yields edit scripts within milliseconds, and these edit scripts are concise, this pipeline can effectively drive incremental computations without significant slowdown.

The new IncA driver crucially relies on the type-safety of edit scripts, because it allows for a more compact data representation. Specifically, we use an encoding of mutable algebraic data:

```
mutable.Map[Link, BidirectionalOneToOneIndex[URI, URI]]
```

This encoding requires a link `Add.e1` to connect an `Add` node to at most one child node in the one-to-one index. With untyped edit scripts, we have to choose a weaker encoding, where a node can have many children for the same link.

```
mutable.Map[Link, BidirectionalManyToOneIndex[URI, URI]]
```

This induces extra performance costs since all operations become set operations now. The code of the new IncA driver is available open source at https://gitlab.rlp.net/plmz/inca-scala.

7 Related Work

Even though Unix *diff* dates back to the seventies, it is at the heart of many modern distributed version control systems, including git, mercurial, and darcs. The Unix *diff* tool compares files on a line-by-line basis, while attempting to share as many lines as possible [10]. While Unix *diff* explains which lines changed, *truediff* explains how two files changed structurally. Asenov et al. [2] show that it is possible to produce structural patches with Unix *diff* by preparing the source file to contain a single AST node per line. The output of Unix *diff* then essentially describes AST node insertions and deletions. Through post-processing of this line-based patch, Asenov et al. can also detect moved nodes. Overall, their approach has quadratic run-time complexity and processing a single Java file can take up to a minute.

Structural diffing was pioneered by Chawathe et al. [4], whose approach we discussed throughout this paper. They first compute a bipartite node matching between source and target tree connecting all nodes that are deemed similar based on heuristics. Then, using matched nodes as anchor points, they derive a provably optimal edit script containing insert, delete, move, and update (of literals) operations. 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, a property approaches based on Chawathe et al. fail to satisfy. By mimicking *truechange*, it may be possible to make the approach by Chawathe et al. type-safe. In particular, it may be possible to generate detach and attach edits instead of move edits, but to use their similarity scores. We have not explored this direction.

The algorithm by Chawathe et al. [4] produces an optimal edit script relative to a given node matching. However, the node matching heavily relies on heuristics, which frequently had to be specialized to obtain satisfactory edit scripts. Indeed, years of research in similarity scores have tried to yield increasingly concise edit scripts:

- Chawathe et al. original heuristics were optimized for flatly structured documents.
- Fluri et al. [7] propose heuristics based on statistical metrics to improve the node matching on generic syntax trees.
- Falleri et al. [6] optimize these heuristics for fine-grained differencing of Java ASTs in the GumTree tool.
- Nguyen et al. [14] tailor the heuristics to clone detection for JavaScript.
- Dotzler and Philippsen [5] propose a number of optimizations applicable to all of the previous approaches, including a move optimization using cryptographic hashes for isomorphic subtrees.

In the design of *truediff*, we do not rely on similarity scores but instead designate reusable trees based on structural and literal subtree equivalences. We showed that *truediff* runs in linear time whereas similarity-based node matching must compare each source node to each target node, yielding a quadratic run-time complexity. Indeed, our empirical comparison demonstrates that using subtree equivalences yields edit scripts that have the same size but can be computed much faster and be type-safe.

The first type-safe structural diffing algorithm was proposed by [12], and this work was later extended to also support polymorphic data types [22]. Both of these approaches compute edit scripts that only reason about insertions, deletions, and updates, but not moves. The computed edit scripts can be interpreted in conjunction with a pre-order tree traversal, where each edit operation applies to the currently visited node. The lack of move edits can significantly increase the length of edit scripts because a moved subtree must be deleted and re-inserted, requiring many edit operations. Therefore, the size of the edit scripts is proportional to the size of the source and target tree.

Our work was inspired by the type-safe structural diffing algorithm hdiff by Miraldo and Swierstra [13]. Like hdiff, we assign cryptographic hashes to subtrees and use these hashes as keys of a hash trie to efficiently compare subtrees. 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. For example, an incremental type checker assigns different types to a variable node, depending on its context. Second, the size of the patch computed by hdiff is proportional to the size of the source and target trees, despite supporting move edits. This is because the generated patch enumerate all nodes leading to a moved (copied) subtree. In contrast, *truediff* uses URIs to identify changed nodes and does not mention unchanged nodes in the patch. And third, the running time of hdiff was unsatisfactory and precluded its application in incremental computing. As our evaluation showed, *truediff* resolves these limitations.

Incremental parsers produce updated ASTs after the source file changed. One may wonder if an incremental parser could not generate an edit script on the side. Indeed, incremental parsing has attracted a lot of attention in the past [16]. Tree-sitter[2] is a modern incremental parsing implementation based on the incremental LR parsing algorithm by Wagner and Graham [24]. This algorithm tries to reuse subtrees of the previous AST, but only if their relative position has not changed (no moves of subtrees). In practice, tree-sitter generates updated AST efficiently, but only reveals which subtrees contain changes, not how they changed. However, the subtrees identified by tree-sitter can be considered an over-approximation of the actual patch. Thus, it would be sound to apply *truediff* on this over-approximation only, which would improve the performance of *truediff* even further.

To the best of our knowledge, our theoretical treatment of edit scripts and the application of linear type systems is novel. However, researchers have proposed formalizations for version control systems. For example, Swierstra and Löh use separation logic to formally reason about patches, conflicts, merging, and branching [18]. They identify basic edit operations for *files* such as the creation of a file, deletion of a file, and atomically updating the file content as a whole. In contrast, we reason about structural edits of file contents. The darcs version control system promotes patches as first-class constructs [17], rather than storing different versions of a file. The patch management of darcs has been formally treated, for example, based on abstract algebra [11] and homotopy type theory [1]. It would be interesting to explore how our fine-grained and structural edit scripts could be leveraged in the formalization of version control systems.

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.

Acknowledgments

We thank Sven Keidel, the anonymous reviewers, and the shepherd for valuable feedback that improved this work.

[2] https://tree-sitter.github.io/tree-sitter/

References

[1] Carlo Angiuli, Edward Morehouse, Daniel R. Licata, and Robert Harper. 2014. Homotopical Patch Theory. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 243–256. https://doi.org/10.1145/2628136.2628158

[2] Dimitar Asenov, Balz Guenat, Peter Müller, and Martin Otth. 2017. Precise Version Control of Trees with Line-Based Version Control Systems. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering - Volume 10202*. Springer-Verlag, Berlin, Heidelberg, 152–169. https://doi.org/10.1007/978-3-662-54494-5_9

[3] Eugene Burmako. 2013. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013*. ACM, 3:1–3:10. https://doi.org/10.1145/2489837.2489840

[4] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change Detection in Hierarchically Structured Information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and Inderpal Singh Mumick (Eds.). ACM Press, 493–504. https://doi.org/10.1145/233269.233366

[5] Georg Dotzler and Michael Philippsen. 2016. Move-optimized source code tree differencing. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 660–671. https://doi.org/10.1145/2970276.2970315

[6] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 313–324. https://doi.org/10.1145/2642937.2642982

[7] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Software Eng.* 33, 11 (2007), 725–743. https://doi.org/10.1109/TSE.2007.70731

[8] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. 2015. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 748–766. https://doi.org/10.1145/2814270.2814305

[9] Daco Harkes, Danny M. Groenewegen, and Eelco Visser. 2016. IceDust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.11

[10] James Wayne Hunt and M Douglas MacIlroy. 1976. *An algorithm for differential file comparison.* Bell Laboratories Murray Hill.

[11] Judah Jacobson. 2009. A formalization of darcs patch theory using inverse semigroups. *Available from ftp://ftp.math.ucla.edu/pub/camreport/cam09-83.pdf* (2009).

[12] Eelco Lempsink, Sean Leather, and Andres Löh. 2009. Type-safe diff for families of datatypes. In *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2009, Edinburgh, United Kingdom, August 31 - September 2, 2009*, Patrik Jansson and Sibylle Schupp (Eds.). ACM, 61–72. https://doi.org/10.1145/1596614.1596624

[13] Victor Cacciari Miraldo and Wouter Swierstra. 2019. An efficient algorithm for type-safe structural diffing. *Proc. ACM Program. Lang.* 3, ICFP (2019), 113:1–113:29. https://doi.org/10.1145/3341717

[14] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H Pham, Jafar Al-Kofahi, and Tien N Nguyen. 2011. Clone management for evolving software. *IEEE transactions on software engineering* 38, 5 (2011), 1008–1026.

[15] André Pacak, Sebastian Erdweg, and Tamás Szabó. 2020. A Systematic Approach to Deriving Incremental Type Checkers. *Proc. ACM Program. Lang.* 4, OOPSLA (2020).

[16] G. Ramalingam and Thomas Reps. 1993. A Categorized Bibliography on Incremental Computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 502–510. https://doi.org/10.1145/158511.158710

[17] David Roundy. 2005. Darcs: Distributed Version Management in Haskell. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell* (Tallinn, Estonia) *(Haskell '05)*. Association for Computing Machinery, New York, NY, USA, 1–4. https://doi.org/10.1145/1088348.1088349

[18] Wouter Swierstra and Andres Löh. 2014. The Semantics of Version Control. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) *(Onward! 2014)*. Association for Computing Machinery, New York, NY, USA, 43–54. https://doi.org/10.1145/2661136.2661137

[19] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *PACMPL* 2, OOPSLA (2018), 139:1–139:29. https://doi.org/10.1145/3276509

[20] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 320–331. https://doi.org/10.1145/2970276.2970298

[21] Tamás Szabó, Edlira Kuci, Matthijs Bijman, Mira Mezini, and Sebastian Erdweg. 2018. Incremental overload resolution in object-oriented programming languages. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*, Julian Dolby, William G. J. Halfond, and Ashish Mishra (Eds.). ACM, 27–33. https://doi.org/10.1145/3236454.3236485

[22] Marco Vassena. 2016. Generic Diff3 for algebraic datatypes. In *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, James Chapman and Wouter Swierstra (Eds.). ACM, 62–71. https://doi.org/10.1145/2976022.2976026

[23] Philip Wadler. 1990. Linear Types can Change the World!. In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, Manfred Broy and Cliff B. Jones (Eds.). North-Holland, 561.

[24] Tim A. Wagner and Susan L. Graham. 1998. Efficient and Flexible Incremental Parsing. *ACM Trans. Program. Lang. Syst.* 20, 5 (Sept. 1998), 980–1013. https://doi.org/10.1145/293677.293678

[25] David Walker. 2005. Substructural type systems. *Advanced topics in types and programming languages* (2005), 3–44.