Grove: A Bidirectionally Typed Collaborative Structure Editor Calculus

MICHAEL D. ADAMS, National University of Singapore, Singapore
ERIC GRIFFIS, University of Michigan, USA
THOMAS J. PORTER, University of Michigan, USA
SUNDARA VISHNU SATISH, University of Michigan, USA
ERIC ZHAO, University of Michigan, USA
CYRUS OMAR, University of Michigan, USA

Version control systems typically rely on a *patch language*, heuristic *patch synthesis algorithms* like `diff`, and *three-way merge algorithms*. Standard patch languages and merge algorithms often fail to identify conflicts correctly when there are multiple edits to one line of code or code is relocated. This paper introduces Grove, a collaborative structure editor calculus that eliminates patch synthesis and three-way merge algorithms entirely. Instead, patches are derived directly from the log of the developer's edit actions and all edits commute, i.e. the repository state forms a commutative replicated data type (CmRDT). To handle conflicts that can arise due to code relocation, the core datatype in Grove is a labeled directed multi-graph with uniquely identified vertices and edges. All edits amount to edge insertion and deletion, with deletion being permanent. To support tree-based editing, we define a decomposition from graphs into *groves*, which are a set of syntax trees with conflicts—including local, relocation, and unicyclic relocation conflicts—represented explicitly using holes and references between trees. Finally, we define a type error localization system for groves that enjoys a *totality* property, i.e. all editor states in Grove are statically meaningful, so developers can use standard editor services while working to resolve these explicitly represented conflicts. The static semantics is defined as a bidirectional marking system in line with recent work, with gradual typing employed to handle situations where errors and conflicts prevent type determination. We then layer on a unification-based type inference system to opportunistically fill type holes and fail gracefully when no solution exists. We mechanize the metatheory of Grove using the Agda theorem prover. We implement these ideas as the *Grove Workbench*, which generates the necessary data structures and algorithms in OCaml given a syntax tree specification.

CCS Concepts: • **Software and its engineering** → *General programming languages*; **Syntax**; **Software configuration management and version control systems**.

Additional Key Words and Phrases: version control systems, collaborative editing, CRDTs, gradual typing

ACM Reference Format:
Michael D. Adams, Eric Griffis, Thomas J. Porter, Sundara Vishnu Satish, Eric Zhao, and Cyrus Omar. 2025. Grove: A Bidirectionally Typed Collaborative Structure Editor Calculus. *Proc. ACM Program. Lang.* 9, POPL, Article 73 (January 2025), 29 pages. https://doi.org/10.1145/3704909

Authors' Contact Information: Michael D. Adams, National University of Singapore, Singapore, Singapore, adamsmd@nus.edu.sg; Eric Griffis, University of Michigan, Ann Arbor, USA, egriffis@umich.edu; Thomas J. Porter, University of Michigan, Ann Arbor, USA, thomasjp@umich.edu; Sundara Vishnu Satish, University of Michigan, Ann Arbor, USA, svishnus@umich.edu; Eric Zhao, University of Michigan, Ann Arbor, USA, zzhaoe@umich.edu; Cyrus Omar, University of Michigan, Ann Arbor, USA, comar@umich.edu.

ACM 2475-1421/2025/1-ART73
https://doi.org/10.1145/3704909

1 Introduction

Development teams typically collaborate with the aid of a version control system (VCS) like Git, Subversion, or Darcs [32]. These systems maintain a branching history of *commits* to a source code repository, each consisting of a *patch* together with various metadata, e.g. a human-readable commit message. Patches are imperative programs written in a *patch language* defining a set of primitive editing commands. The standard POSIX `patch` language, for example, specifies commands for inserting and deleting specified lines of text at specified line numbers within a file.

Developers do not typically express program edits using the patch language directly, nor in any case do version control systems typically have access to a log of the developer's edits. Instead, version control systems must *synthesize* patches from the file system state using heuristic algorithms, such as the classic `diff` algorithm that synthesizes a patch that minimizes the edit distance between two file system states [14].

When two patches, developed concurrently in branches based on a common ancestral commit, must be merged, version control systems deploy a *three-way merge algorithm*. The standard approach is to apply the *local patch* first, then modify the *remote patch* by shifting its line numbers to account for the local patch's line insertions and deletions. This algorithm is an *operational transform* [9]. Character-level operational transforms are similarly used for real-time collaborative editing, e.g. in tools like Google Docs and in Visual Studio Code's Live Share feature.

Problem 1: Merging Sensibly. When merging patches, conflicts (e.g. due to different modifications to the same location) are unavoidable. However, standard three-way merge algorithms for line-based patch languages commonly identify spurious conflicts, fail to identify legitimate conflicts, or silently duplicate or misplace code. Let us briefly review some classic problems with these systems. The supplemental material includes Git repositories that demonstrate each of these problems.

The **granularity problem** arises when both commits make edits to different locations within a single line of code. For example, one patch might rename a function argument while the other adds a new argument. Similarly, if one patch renames a type while another makes unrelated changes to code that references that type, there will be conflicts at every line of code shared between the two patches. A notable special case is the **nesting problem**, which arises when one patch changes the nesting of code structures, e.g. by wrapping a code block within a new control flow construct, thereby changing the indentation of every line in the block. These changes can cause conflicts.

The **relocation conflict problem** arises when two patches relocate a code block to different locations. Standard patch languages operationalize code relocation as simply a deletion paired with an insertion. The merge will therefore fail to identify this legitimate location conflict and instead silently duplicate the code block at both locations.

The **relocation modification problem** arises when one commit relocates a code block that another commit modifies. A naive approach would silently leave the modifications at the original location. A more sophisticated block-based approach, like that deployed by `git`, might indicate a conflict when an insertion occurs within the bounds of a code block that has been deleted.

These classic problems have motivated research into richer patch languages, more sophisticated patch synthesis algorithms, and corresponding improvements to three-way merge algorithms. For example, systems like Git address the nesting problem by allowing indentation changes from one patch to be merged with other changes that do not modify indentation. More sophisticated systems deploy parsers and tree differencing algorithms [6, 10, 11, 19, 23, 34] to better address the granularity problem.

Addressing the relocation-related problems is more difficult. A common approach is to enrich the patch language to make code relocation a primitive command. However, correctly synthesizing code relocation commands given only the initial and final states of the repository requires heuristics.

The typical approach is to assume that a matching deletion and insertion is due to code relocation. However, relocated code is often also modified. In these cases, it is difficult to determine whether a deletion and a similar but not identical insertion are related by relocation, rather than coincidental code similarity. Developers often intentionally copy and then modify code, so there may be multiple partially matching insertions for a given deletion and there is no clear way to decide which, if any, are related by relocation. The developer's actual actions, e.g. cuts, copies, and pastes, are not persisted into the file system, nor are there persistent identifiers associated with code structures represented as text, so text-file-based systems have no choice but to deploy imperfect heuristics.

Contribution 1: Grove: A Collaborative Structure Editor Calculus. This paper considers the problem of collaborative editing for *structure editors*, which eschew text editing. Instead, developers code by applying tree edit actions directly to a continuously evolving *program sketch*, i.e. a syntax tree with holes, shown projected visually in various ways to the developer. Structure editing has been studied since the 1980s with the Cornell Program Synthesizer [39] and research continues to this day, with numerous active projects including Scratch [21] and other block-based editors (which are widely used in educational and end-user programming settings), Jetbrains MPS (which has been deployed in industry) [40], and Hazel (a live functional programming environment rooted in a structure editor calculus called Hazelnut, which serves as an active research platform) [25].

Inspired by Hazelnut, this paper introduces Grove, a *collaborative structure editor calculus* for arbitrary syntax trees that does not suffer from the problems just outlined. This is in large part due to a substantial simplification of the overall collaborative editing architecture. In particular, we *eliminate patch synthesis (i.e. diff algorithms) entirely*, instead deriving patches directly from the log of edits performed by the developer. We also *eliminate the need for three-way merge algorithms (i.e. operational transforms) entirely.* Instead, we define the patch language such that all edits commute, so remote patches can be applied without transformation. We prove a *convergence theorem* that ensures that branches of a repository will converge to the same state when the same set of patches are applied, regardless of the order in which they are applied. The patch language forms what is known as a *commutative replicated data type (CmRDT)* [29, 35].

Defining a structure editor calculus that supports code insertion, deletion, and relocation using only commutative edits, and avoiding the problems outlined above, is not trivial. The Hazelnut action language is neither commutative nor does it support relocation. Relocation is particularly challenging because of the potential for relocation conflicts, including the potential for cyclic relocation (when one commit relocates node A beneath node B, and the other *vice versa*).

This need to represent conflicting states means that we cannot use a single syntax tree as Grove's core data structure. Instead, we use a directed labeled multi-graph. A vertex corresponds to a syntax tree node and is labeled with a *unique identifier (UID)* and a *constructor*, e.g. `Plus`. An edge is also labeled with a UID and establishes a parent-child relationship at a labeled *position* for the parent vertex's constructor, e.g. at the `L` or `R` position of the `Plus` constructor. We refer to a parent vertex and position collectively as a *location*.

The *Grove patch language* is quite simple: a patch can insert or delete an edge (which might cause the creation of a mentioned vertex if it did not already exist). To ensure commutativity, deletion of an edge is permanent, i.e. the edges form a two-phase set (2P-Set) CmRDT [35]. The edit actions that users perform are given meaning by a straightforward translation to a graph patch. Relocation simply translates to edge deletion and insertion. Critically, vertices are *not* deleted during relocation, so we do not need to deploy heuristics to identify relocated nodes.

Structure editors are designed to provide editing affordances for trees, not graphs where there may be any number of children at a given location, vertices may have any number of parents, and where there may be cycles. To support conventional tree-based structure editing, we define

a *decomposition* of our program graph into a *grove*, which is a set of programs with holes, local conflicts, and conflict references between them to account for motifs that arise when editing collaboratively. In particular, a location with no out-edges decomposes to a *hole*. More than one out-edge at a given location decomposes to an explicitly represented *local conflict*. More than one in-edge indicates that a vertex has a *relocation conflict*, so we leave a *relocation conflict reference* at each of the conflicting locations. Finally, cycles are broken during decomposition by leaving a *unicycle conflict reference* at an arbitrary (but deterministically chosen) edge. Resolving these various conflicts simply requires manipulating these constructs like any other syntactic construct, e.g. deleting all but one relocation conflict reference to determine a unique location for a node.

Problem 2: Semantic Gaps During Conflict Resolution. When working with traditional version control systems, resolving conflicts can take time and require reasoning about syntax, types, and program behavior. Traditionally, however, conflicts are indicated by inserting extra-linguistic markers into files. These markers are not typically understood by the parser, and because they include conflicting alternatives, they cannot generally be removed or concatenated to result in a sensible program. Consequently, language services that require a well-formed, meaningful program (e.g. type error localization, go-to-definition, live evaluation and so on) either fail to operate or exhibit gaps in service, e.g. because they are relying on data from a compile prior to the merge attempt. Developers are left to reason without the aid of much of their tooling during conflict resolution. This is an instance of the more general **semantic gap problem** when programming tools encounter incomplete programs [26].

The previous work on the Hazelnut structure editor calculus addresses the semantic gap problem in the single-user setting by defining a type system and type error localization system (the *marked lambda calculus*) for incomplete programs, i.e. programs with holes [43]. Notably, type error localization is proven to be *total*, i.e. the system is able to assign static meaning to *every* syntactically well-formed expression by inserting *marks* to localize errors. Marking employs local type inference as codified by a *bidirectional type system* [8] as well as *gradual typing* [36], i.e. the theory of *type holes*, to recover from situations where type errors make it impossible to determine a known type. A separate type hole filling phase deploys unification-based (i.e. non-local) type inference to fill type holes when possible, or allows the user to interactively select from hole fillings that partially satisfy generated constraints when there are type conflicts.

Contribution 2: Total Type Error Localization and Recovery for Groves. This paper extends this prior work on the marked lambda calculus to develop a total type error localization system for groves, introduced in Contribution 1 above as the result of decomposing a commutatively edited graph into a set of terms with empty holes, local conflicts, relocation conflict references, and unicycle conflict references. This paper develops a type (and type error localization) discipline for handling these novel constructs. We follow the marked lambda calculus in rooting our *marked grove calculus* in bidirectional type checking, deploying gradual typing when conflicts do not allow a single type to be inferred, and then layering on a unification-based type inference system to opportunistically fill holes or suggest partial solutions when there are conflicting types due to conflicting syntax.

Paper Outline. Section 2 introduces Grove by example, demonstrating its behavior in each of the problematic scenarios named above. Section 3 then formally defines Grove's graph structure, commutative patch language, grove decomposition procedure, and edit action language. We establish key metatheoretic properties using the Agda proof assistant. Section 4 describes our implementation of the Grove Workbench, which is defined modularly to allow it to be instantiated with arbitrary syntax trees. Section 5 instantiates Grove with a simply typed lambda calculus, then defines a bidirectional type and type localization system for groves and proves totality and other key

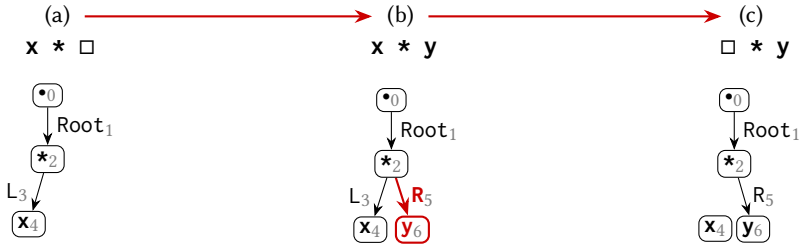


Fig. 1. (a) We represent collaborative program sketches as graphs. (b) Hole filling translates to edge insertion. (c) Term deletion (or cut) deletes an edge, but the vertex persists. (We omit it in subsequent figures.)

metatheoretic properties using the Agda proof assistant. Section 6 reviews related work in more detail. Section 7 concludes with a discussion and directions for future work on collaborative editing.

2 Grove By Example

This section introduces collaborative structure editing in Grove by example. Section 2.1 describes how we use graphs to represent collaborative program sketches. Section 2.2 then shows examples of edits being performed by a single user, Alice. Sections 2.3–2.4 then describe a collaboration between two users, Alice and Bob, as they edit their own branches of a program and periodically merge in each other's edits, starting with examples without conflicts, then considering the various kinds of conflicts that might arise.

For simplicity and concision, all of the examples in this section will be for a language of standard arithmetic operations, but our formalism in Section 3 and our implementation in Section 4 are parameterized by an arbitrary abstract syntax.

Grove can form the basis for both a conventional version control workflow, where edits are batched into commits, or real-time collaborative editing, where edits are communicated as they occur. This paper makes no assumptions about which batching mode is in use, nor do we consider the well-studied problem of reliably and efficiently communicating patches over a network.

2.1 Representing Collaborative Program Sketches as Graphs

The *edit state* of a Grove branch is a directed multi-graph representing a *collaborative program sketch*, meaning an incomplete program, i.e. one that may have *holes* and (as we will return to) conflicts. For example, Figure 1a gives one such graph and its corresponding *decomposition* into, in this case, a single syntax tree, `x * □`, whose missing right operand is a hole, denoted □.

Each vertex represents a term in the specified language, except for a distinguished root vertex, and is labeled with a unique identifier (UID) and a *constructor*. In Figure 1a, the root vertex has UID 0 and constructor •. Vertices $*_2$ and $\mathtt{x}_4$ have UIDs 2 and 4 and constructors `*` and `var(`x`)`, respectively. For clarity, we abbreviate `var(`x`)` as simply `x`; here, x is a constructor parameter. We treat identifiers and literals as indivisible, but we discuss character-level editing in Section 7.

An edge indicates that the destination vertex is a child of the origin vertex. Each edge is labeled with a UID (e.g., 1 and 3 in Figure 1a) and a *position* (e.g., `Root` and `L` in Figure 1a). The parent vertex's constructor determines a set of valid positions. For instance, the `*` constructor defines positions `L` (for the left operand) and `R` (for the right operand). The `var` constructor is a leaf so it defines no positions. The root vertex constructor • has a single child position, `Root`.

Holes arise in the decomposition by the absence of a child at a valid position. For example, in Figure 1a the absence of an `R` child under $*_2$ corresponds to the hole in the right operand of `x * □`.

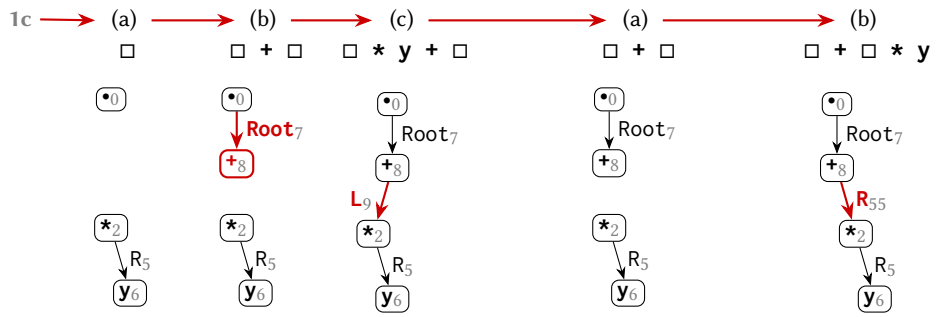


Fig. 2. Wrapping results in a patch which (a) cuts the original term (edge deletion), (b) creates the outer term at the same location (edge insertion), then (c) pastes the original term (edge insertion).

Fig. 3. Term relocation results in a patch which (a) cuts the term (edge deletion) then (b) pastes it in its new location (edge insertion). Vertex identities and downstream edges are conserved.

For clarity, we use even numbers for vertex UIDs and odd numbers for edge UIDs. In practice, UIDs would be generated by a mechanism that effectively ensures that collaborators always generate distinct UIDs, e.g. by generating universally unique IDs (UUIDs) [28].

2.2 Structure Editing

Individual users perform *edits* to evolve the edit state. We consider several standard edits, including insertion, deletion, cut-and-paste (relocation), copy-and-paste, and undo/redo. This paper abstracts over the user interface aspects of structure editors and makes no usability-related claims; these edits could be performed through, for example, drag-and-drop interactions (as in block-based editors like Scratch) or keyboard interactions (as in MPS and Hazel).

Each edit translates directly to a *graph patch*, which consists of a sequence of *patch commands*. The Grove patch language requires only two patch commands: *edge insertion* and *edge deletion*. A vertex is inserted when it is included in an edge insertion command.

To illustrate the Grove patch language, let us consider a sequence of standard edits, found across structure editors, performed by a single user, Alice.

2.2.1 Hole Filling. First, Alice fills the hole in the right position of `x * □` from Figure 1a with the variable `y`. The resulting edit state is shown in Figure 1b. The patch corresponding to this hole filling action inserts an edge, labeled R_5, from the vertex corresponding to the parent term, $*_2$, to the newly constructed variable's vertex, y_6. The resulting graph decomposes to the term `x * y`.

2.2.2 Deletion. Next, Alice moves the cursor to `x` in Figure 1b and deletes it, causing the deletion of edge L_3 and resulting in the decomposition `□ * y` as shown in Figure 1c.

Once an edge with a particular identifier is deleted, it cannot be re-inserted. For instance, if Alice performed an "undo" on this deletion, a fresh edge between $*_2$ and x_4 would be created. (We can allow simple undo only if the patch has not yet been communicated to a collaborator).

Notice that vertex x_4 continues to exist (and if it had any children, they would remain connected to it; see below for the implications in the collaborative setting). In the remaining figures, we omit such orphaned vertices if they are not relevant to the exposition.

2.2.3 Wrapping. Next, Alice moves the cursor to the parent term $*_2$ in Figure 1c with the intention of wrapping it in a binary addition expression with constructor `+`.

Many structure editors define a primitive wrapping edit, choosing a position heuristically (e.g. favoring the left). Others require the user to cut the original term, construct the new outer term, then paste the original term in the intended position.

In either case, the corresponding sequence of patch commands would produce the edit states shown in Figure 2: the edge connecting the root to the original term is deleted (effectively cutting the original term), leaving $\star_2$ temporarily orphaned, then an edge to the new outer term is inserted, followed by an edge reconnecting the original term (effectively pasting the original term).

2.2.4 Relocation. Alice changes her mind and decides to relocate the multiplication from the left to the right position of the addition. A structure editor might support this using drag-and-drop or cut-and-paste affordances. In either case, the resulting patch commands will proceed through the two states shown in Figure 3: deleting the original incoming edge and then inserting an edge at the new location. Notice that the sub-graph corresponding to the relocated term itself is never deleted nor re-inserted, in contrast to conventional line-based patch languages. See below for the implications in the collaborative setting.

2.2.5 Copying. A copy-and-paste, or a cut followed by multiple pastes, would of course involve copying the graph structure of the original term but generating fresh UIDs (not shown).

2.3 Collaboration

We now turn our attention to how Grove handles collaboration. The examples in this section generalize to collaborations between any number of users, but for simplicity we consider only two: Alice and Bob. Alice and Bob are each concurrently editing their own branches of the repository (or their own instance of a real-time collaborative editor), performing edits that translate to patches as described above. They periodically communicate these patches to one another. Figure 4 diagrams the Grove collaboration model.

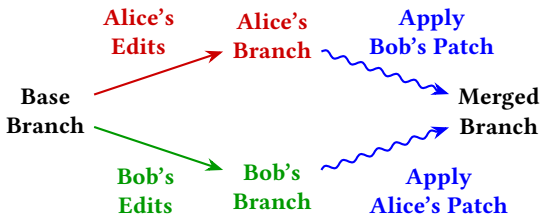


Fig. 4. Collaboration in Grove is simple due to the commutativity of Grove's patch language.

2.3.1 Commutativity. The Grove patch language is commutative, meaning that there is no need for a complex three-way merge algorithm (i.e. operational transform). Instead, each user can simply apply incoming patches to their own edit state as they arrive, no matter the order in which they arrive. If two users have received the same set of patches, their edit state will converge.

The key properties that make the Grove patch language commutative is that edge deletion is permanent and vertex insertion is permanent. We establish commutativity formally in Section 3. For now, let us consider several example scenarios that demonstrate how Grove handles different collaborative editing scenarios, particularly the problematic situations outlined in Section 1.

2.3.2 Solving the Granularity Problem. Alice and Bob start where Alice left off in Figure 3b with the term $\square + \square * y$. Alice then adds u_{12} as the left child of $\star_2$. Concurrently, Bob changes y_6 to v_{14}. Before sharing their patches, Alice and Bob have the edit states Figure 5a and Figure 5b,

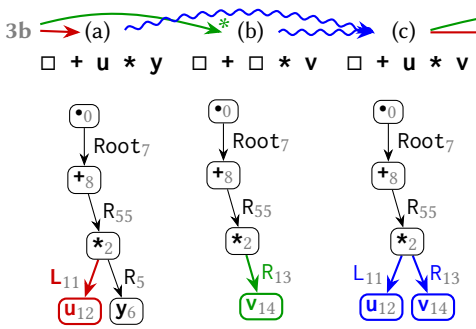


Fig. 5. (a) Alice fills the hole in the multiplication with u. (b) Bob renames y to v. (c) These edits to the same expression commute, addressing the **granularity problem**.

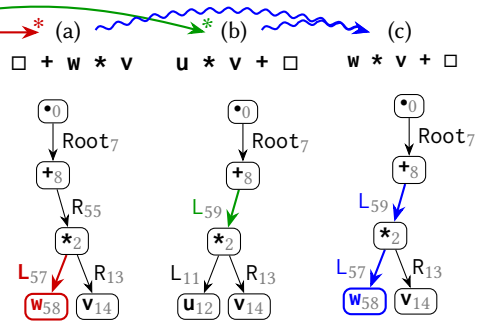


Fig. 6. (a) Alice renames u to w. (b) Bob relocates the multiplication to the left side. (c) These edits commute without conflict or duplication, addressing the **relocation modification problem**.

respectively. Note that the transition from Figure 3b to Figure 5b represents multiple graph updates, i.e., deleting R_5 and adding R_{13} along with its child v_{14}. We thus mark the transition with a star. Once Alice and Bob share their patches and apply each other's patch to their own edit state, both edit states converge to the graph in Figure 5c. Because Grove's patch language is structural rather than line-based, the fact that these edits happened to be close to one another (i.e. in the same arithmetic expression) does not run afoul of the **granularity problem** described in Section 1.

2.3.3 Solving the Relocation Modification Problem. After converging on □ + u * v in Figure 5c, Alice changes u_{12} to w_{58}, producing the edit state in Figure 6a. Meanwhile, Bob relocates $*_2$ (bringing along its children) from the R position of $+_8$ to the L position. As discussed above, this involves deleting R_{55} and adding L_{59}. Bob's resulting edit state is shown in Figure 6b.

Although Alice has modified a term that Bob has concurrently relocated, the edits commute: Alice's modifications are relocated to the new location chosen by Bob. This addresses the **relocation modification problem** described in Section 1. In a line-based setting, this kind of edit can lead to silent code duplication or spurious conflicts (the threat of which, in the author's experience, can inhibit development teams from performing useful code reorganizations).

2.3.4 Warning of Edits under Disconnected Terms. If instead of relocating the multiplication in Figure 6, Bob had deleted it (i.e. disconnected it from the root), then Alice's edits would still commute, but her edits would be applied under a deleted term.

This situation could also arise in a real-time collaborative editor, where each individual edit might arrive at any time (rather than in atomic commits). If Alice, say, receives Bob's deletion of R_{55}, then makes her edits before receiving Bob's subsequent insertion of L_{59} to complete the relocation, Alice's edits would then temporarily be under a disconnected term.

This does not present a formal problem or conflict. A subsequent edit might reconnect a disconnected term, so it is sensible for edits to these terms to be recorded. However, heuristically, a system might warn users, perhaps after a period of quiescence in a real-time setting, that Alice's edits were effectively deleted and provide affordances for interacting with disconnected terms.

2.4 Conflicts

The collaborative edits discussed so far merge cleanly, but in general, merging patches can lead to graphs that do not map cleanly to a conventional syntax tree. We identify several different motifs

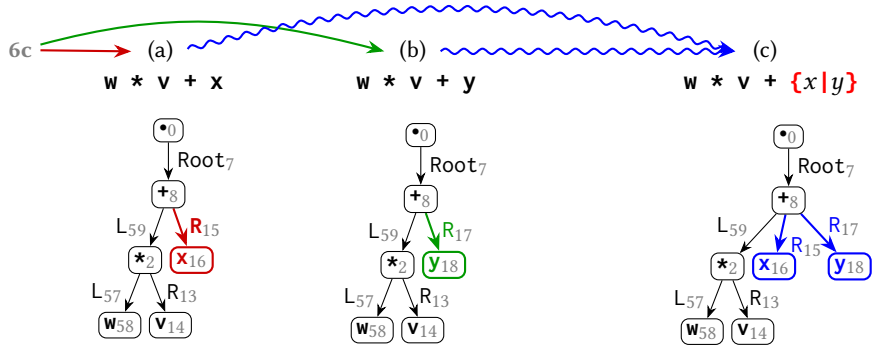


Fig. 7. (a) Alice fills the hole at the R position of the addition with x. (b) Bob fills the same hole with y. (c) When the corresponding patches are merged, there are two edges in the R position of the addition. Decomposition turns these into a *local conflict*, leaving it to the users to resolve the problem by performing normal edits.

that might arise, all of which give rise to different kinds of conflicts in the graph decomposition: *local conflicts*, *relocation conflicts*, and *unicyclic relocation conflicts*. As with merge conflicts in version-control systems such as git, these all require user intervention to resolve.

2.4.1 Local Conflicts. Suppose Alice and Bob both start with the edit state `w * v + □` from Figure 6c. Alice moves the cursor to the hole and constructs x_{16} as the R child of $+_8$. At the same time, Bob constructs y_{18} at the same location. Now Alice and Bob have the graphs in Figure 7a and Figure 7b, respectively.

When these patches are merged in Figure 7c, *both* R_{15} and R_{17} appear in the merged graph. When decomposing this graph to a syntax tree, we resolve this conflict in the R position of $+_8$ by decomposing to a *local conflict*, { x | y }.

Local conflicts can be resolved simply by deleting or relocating all but one of the conflicting terms (and editing the remaining term into the correctly merged value, if needed), which would remove the corresponding edge. For example, Alice could resolve the problem by wrapping x and y with a multiplication, effectively moving them to non-conflicting locations. No special edit actions are needed for conflict resolution.

It is worth noting one special situation: if Alice and Bob independently filled the hole with structurally identical terms, e.g. x, Grove would *still* formally identify a conflict, because the terms have distinct UIDs. In this situation, it would be reasonable for the system to resolve the conflict without further coordination by deterministically choosing one of the two terms, e.g. the smallest.

When the conflicted terms are similar up to UID differences but not identical, it might be helpful to give the developer the option to "push down" the conflicts as deeply as possible, using a tree differencing algorithm. However, this would increase the number of conflicts overall, so it may not always be preferable. Tree differencing is not fundamental to the collaboration model of Grove.

2.4.2 Relocation Conflicts. Grove's support for code relocation creates the possibility for *relocation conflicts*. These occur when a merge causes a vertex to have multiple incoming edges, indicating that it does not have a uniquely determined location (as opposed to local conflicts, which occur when there are multiple outgoing edges at a specified location).

For example, Figure 8 shows an example where Alice and Bob relocate a term, w, to two different locations (the two holes in Figure 8a). In both cases, the edits are modeled as one edge deletion followed by one edge addition. Alice deletes L_{57} and adds R_{19} in Figure 8b. At the same time, Bob deletes L_{57} and adds R_{21} in Figure 8c. Edge deletion is idempotent, so the fact that Alice and Bob both

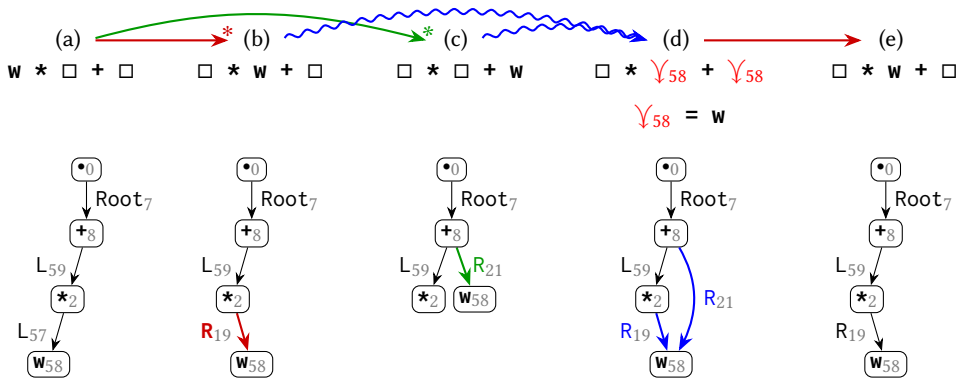


Fig. 8. (a) We start in a state with a variable, w, and two holes. (b) Alice relocates w to the left hole. (c) Bob relocates w to the right hole. (d) After merging, vertex w_{58} has two incoming edges, i.e. it has a *relocation conflict*. The corresponding decomposition leaves a *relocation conflict reference* at both locations, partially addressing the **relocation conflict problem**. Terms that have a relocation conflict are tracked separately by decomposition. (e) The relocation conflict can be resolved by deleting all but one reference.

deleted L_{57} will not lead to a conflict. However, both R_{19} (added by Alice) and R_{21} (added by Bob) point to the same vertex. Once these patches are merged, the resulting edit state is given in Figure 8d. Notice w_{58} has two edges pointing to it. When decomposing the graph, we leave a *relocation conflict reference*, $\downarrow_{18}$, at each conflicting location. The conflicted term is separately tracked in the result of decomposition, which, due to conflicts like these, is formally a set of terms with references between them. We call this set a *grove*. This approach partially addresses the **relocation conflict problem** from Section 1 (we also need to handle unicycles, see below, to fully address the problem).

To resolve relocation conflicts, a user can simply delete all but one of the relocation conflict references. This will cause the corresponding edges to be deleted, and when only one edge remains, there will no longer be a conflict. An editor might provide a more convenient way of deleting all but a selected relocation conflict reference, and provide affordances for displaying these terms, e.g. by transcluding them inline at each location or showing them in a separate sidebar.

2.4.3 Cycles. Relocation in a collaborative setting can also lead to cycles in the graph. For example, consider the situation in Figure 9. Starting in Figure 9a, Alice relocates $*_{24}$ to the R child of $+_8$ and then $+_{26}$ underneath that in Figure 9b. Bob does the opposite, putting $*_{24}$ under $+_{26}$ in Figure 9c. On their own, neither of these edits creates a cycle. However, merging the two patches results in the graph in Figure 9d, which has a cycle between $*_{24}$ and $+_{26}$.

The main difficulty with cycles has to do with decomposition back to a term, i.e. a syntax tree. We do not want decomposition to traverse endlessly attempting to create an infinite tree, so we need to break the cycle somewhere.

If the cycle is connected to a larger term, then there will necessarily be at least one vertex along the cycle that has multiple incoming edges. In Figure 9, both $*_{24}$ and $+_{26}$ have multiple incoming edges. As described above, decomposition will leave relocation conflict references in these positions, thereby breaking the cycle. In this example, these references appear within a local conflict as well, because both vertices were relocated under a common parent vertex. As before, this cycle can be broken by deleting or otherwise modifying the terms until there are no longer any such conflicts.

It is also possible to merge patches such that a *disconnected unicycle* emerges in the graph, even when neither patch disconnects any vertex from the root. Figure 10 shows a simple example of when this could occur: Alice relocates $*_{24}$ under $*_2$, while Bob relocates $*_2$ under $*_{24}$. This results in

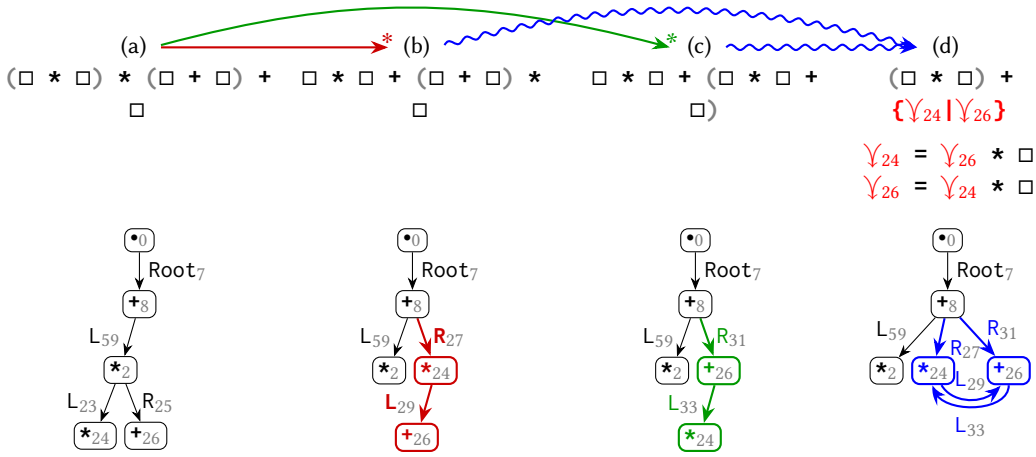


Fig. 9. (a) We start with a tree with a multiplication, $*_{24}$, and addition, $+_{26}$, at the leaves. (b) Alice relocates them both, such that $*_{24}$ is the parent of $+_{26}$. (c) Bob relocates them both, such that $+_{26}$ is the parent of the $*_{24}$. (d) In the merged state, there is a cycle in the graph. Because the terms have a common parent, there is a local conflict. Because the cycle is connected to the rest of the graph, the cycle is broken during decomposition by relocation conflict references as shown.

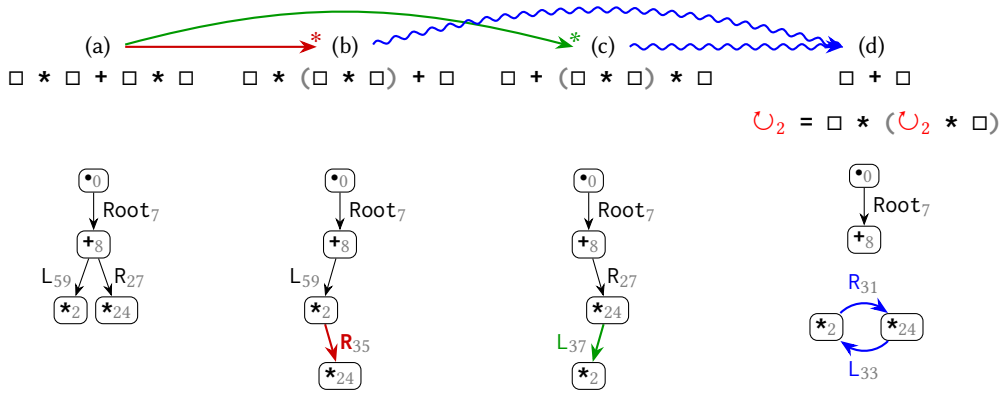


Fig. 10. (a) We start with a tree containing $*_2$ and $*_{24}$. (b) Alice relocates $*_{24}$ to a child position of $*_2$. (c) Bob relocates $*_2$ to a child position of $*_{24}$. (d) When merged, the resulting cycle becomes disconnected from the rest of the graph, forming a *unicycle*. During decomposition, we break unicycles by leaving a *unicycle conflict reference* at an arbitrarily chosen edge.

Figure 10b and Figure 10c, respectively. Merging these causes both vertices to become disconnected, because they were both relocated. The inserted edges form a *unicycle*, meaning a cycle where every vertex has in-degree 1. In this case, we cannot rely on relocation conflict references to break the cycle. Instead, we break the cycle by arbitrarily but deterministically choosing an edge along the unicycle, e.g. the edge with the smallest UID, and leaving a *unicycle conflict reference* at that location, as shown in Figure 10d. A user can be notified of this situation when merging and resolve these conflicts again by relocating or deleting terms until the cycle no longer exists in the graph.

Relocation conflict references and unicycle conflict references together fully address the **relocation conflict problem** from Section 1.

3 The Grove Calculus

We now formalize the ideas presented informally above by defining the Grove Calculus. The calculus consists of a commutative patch language on graphs, defined in Section 3.1, a decomposition procedure to go from graphs to groves (i.e. sets of terms with conflicts) in Section 3.2, and a user action language that operates on groves and is defined by translation to the patch language in Section 3.3.

3.1 A Commutative Graph Patch Language

3.1.1 Graphs. Let $\mathcal{U}$ and $\mathcal{W}$ be separate sets of *unique identifiers* (UIDs) for vertices and edges, respectively. Assume each is equipped with a total ordering $\leqslant$.

Let $\mathcal{K}$ be the set of *constructors* of terms in the source language, and let $\mathcal{P}$ be the set of *positions* of subterms in the source language. Assume each constructor $k \in \mathcal{K}$ is associated with a finite set of positions $\mathsf{arity}(k) \subseteq \mathcal{P}$, representing the set of positions that children of k can inhabit.

For example, if the language is the simply typed lambda calculus, $\mathcal{K}$ will include constructors for function abstractions ($\textsc{Lambda}_x$), function applications (Ap), variables ($\textsc{Var}_x$), and constants ($\textsc{Const}_c$), as well as function type constructors (Arrow) and base types ($\textsc{Base}_b$). The arity of each constructor will reflect the child positions of these constructors, e.g. $\mathsf{arity}(\textsc{Ap}) = \{\textsc{Function}, \textsc{Argument}\}$, $\mathsf{arity}(\textsc{Lambda}_x) = \{\textsc{Annotation}, \textsc{Body}\}$, and $\mathsf{arity}(\textsc{Var}_x) = \varnothing$.

A *vertex* $v = (u, k) \in \mathcal{V} = \mathcal{U} \times \mathcal{K}$ is labeled with a UID, u, and a constructor, k.

A *location* $\ell = (v, p) \in \mathcal{L}$ represents the origin of an edge, where $v = (u, k)$ is drawn from $\mathcal{V}$ and position p is drawn from $\mathsf{arity}(k)$.

An *edge* $\varepsilon = (w, \ell, v) \in \mathcal{E} = \mathcal{W} \times \mathcal{L} \times \mathcal{V}$ represents a directed multi-edge identified by w, originating from location ℓ, with destination vertex v.

A *graph* $g : \mathcal{E} \to \Sigma$ is a function from edges to *edge states* $s \in \Sigma = \{\bot, \boxplus, \boxminus\}$. If $g(\varepsilon) = \bot$, then ε has not yet been constructed in the graph. If $g(\varepsilon) = \boxplus$ then ε is live in the graph. If $g(\varepsilon) = \boxminus$, then ε has been deleted and cannot be constructed again. The total ordering $\bot \sqsubset \boxplus \sqsubset \boxminus$ forms a lattice over Σ. The state of each edge can only progress along this ordering over time, from not yet constructed, to live, to deleted. The *realized* edges in a graph are those assigned to $\boxplus$ or $\boxminus$. We assume that there are finitely many realized edges in a graph.

3.1.2 Graph Patch Commands. We define a graph patch command $\pi = (s, \varepsilon) \in \{\boxplus, \boxminus\} \times \mathcal{E}$ as a pair of an edge state (excluding $\bot$) and an edge. As a shorthand, we use $+\varepsilon$ to denote the construction command $(\boxplus, \varepsilon)$ and $-\varepsilon$ to denote the destruction command $(\boxminus, \varepsilon)$.

For $s_1, s_2 \in \Sigma$, we define the join operation $s_1 \sqcup s_2$ to be the least upper bound of s_1 and s_2 with respect to the $\sqsubset$ ordering. Accordingly, $\boxplus \sqcup \bot = \boxplus$ means that an edge can be created if it has never existed, and $\boxplus \sqcup \boxminus = \boxminus$ means that once an edge is deleted it can never be restored.

We define the semantics of patch commands via the following transition relation between graphs.

$$g \xrightarrow{(s,\varepsilon)} g\left[\varepsilon \mapsto s \sqcup g(\varepsilon)\right]$$

Applying patch command $\pi = (s, \varepsilon)$ to graph g results in the updated graph $g' = g\left[\varepsilon \mapsto s \sqcup g(\varepsilon)\right]$, wherein the edge state associated with edge ε in g becomes the join of s with the state of ε in g, and $g'(\varepsilon') = g(\varepsilon')$ for any $\varepsilon' \neq \varepsilon$. Essentially, these patch command semantics act as a transition system between graphs, *joining* the new edge state with the corresponding edge state of the graph to which the patch command is being applied. Since a patch command can only change the value of g on one input, applying the patch command preserves the property that g maps all but finitely many edges to $\bot$.

A *graph patch* $\bar{\pi}$ is a sequence of graph patch commands, and the transition relation is extended to patches as the composition of the transitions of the constituent patch commands.

$$
\begin{array}{lrcl}
t \in & \mathit{Term} & ::= & k^u\ \{\bar{t}_p\}_{p \in arity(k)} \mid \curlyvee_{(w,v)} \mid \circlearrowright_{(w,v)} \\
\bar{t} \in & \mathit{ChildTerm} & ::= & {}^{\ell}\square \mid {}^{w}t \mid {}^{\ell}\{{}^{w_i}t_i\}_{i<n}
\end{array}
$$

Fig. 11. Syntax of terms

3.1.3 Commutativity. Using these definitions, commutativity of patch commands can be established using the commutativity and associativity of the join operation on edge states.

Lemma 3.1 (Join Semilattice). *$(\Sigma, \sqsubset)$ with $\sqcup$ forms a join semilattice. That is, $\sqcup$ is associative, commutative, and idempotent.*

Proof. Follows directly from the total ordering $(\Sigma, \sqsubset)$. □

Theorem 3.2 (Commutativity). *For all graphs g and g' and all graph patch commands $\pi_1 = (s_1, \varepsilon_1)$ and $\pi_2 = (s_2, \varepsilon_2)$, if $g \xrightarrow{\pi_1\pi_2} g'$, then $g \xrightarrow{\pi_2\pi_1} g'$*

Proof. If $g \xrightarrow{\pi_1\pi_2} g'$, then $g' = g[\varepsilon_1 \mapsto s_1 \sqcup g(\varepsilon_1)][\varepsilon_2 \mapsto s_2 \sqcup g(\varepsilon_2)]$. We want to show that this equals $g'' = g[\varepsilon_1 \mapsto s_1 \sqcup g(\varepsilon_1)][\varepsilon_2 \mapsto s_2 \sqcup g(\varepsilon_2)]$. For any ε, if $\varepsilon \neq \varepsilon_1$ and $\varepsilon \neq \varepsilon_2$, then $g'(\varepsilon) = g''(\varepsilon) = g(\varepsilon)$. If $\varepsilon = \varepsilon_1$ but $\varepsilon \neq \varepsilon_2$, then $g'(\varepsilon) = g''(\varepsilon) = s_1 \sqcup g(\varepsilon)$. Similarly if $\varepsilon = \varepsilon_2$ but not ε_1. Finally, if $\varepsilon = \varepsilon_1 = \varepsilon_2$, then $g'(\varepsilon) = s_2 \sqcup (s_1 \sqcup g(\varepsilon))$, and $g''(\varepsilon) = s_1 \sqcup (s_2 \sqcup g(\varepsilon))$. These are equal by Lemma 3.1. Since $g'(\varepsilon) = g''(\varepsilon)$ for all ε, $g' = g''$ and $g \xrightarrow{\pi_2\pi_1} g'$. □

The commutativity of patch commands generalizes directly to the commutativity of patches.

3.1.4 Interpretation as a CmRDT. The definition above can be understood operationally as a CmRDT. In particular, the edges can be understood as forming a two-phase set (2P-Set), because deletion is permanent [35]. The set of realized edges forms a grow-only set, as does the set of realized vertices (those included in a realized edge).

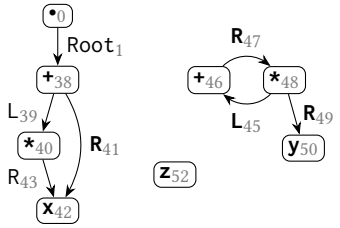
3.2 Groves

Structure editors and other tools operate on trees, not on arbitrary graphs. When performing these operations on a graph g, only the set of edges ε such that $g(\varepsilon) = \boxplus$ should be considered. We call this the *live subgraph* of g.

We present *groves*, a tree-based representation of a live subgraph. A grove is a collection of mutually referential terms annotated with vertex and edge UIDs governed by the grammar in Figure 11. Groves correspond precisely to live subgraphs in the sense that a live subgraph can be *decomposed* into a grove, and the grove can be *recomposed* into the original live graph.

3.2.1 Terms. Given a source language parameterized by a set of constructors $k \in \mathcal{K}$, we define an augmented term language as specified in Figure 11. Terms are extended with cases for relocation conflict references and unicycle conflict references. Additionally, since any position of any constructor may have zero, one, or multiple outgoing edges, we define a secondary sort *ChildTerm* to represent these possibilities. A location with no outgoing edges corresponds to an empty hole, a location with one outgoing edge corresponds to an ordinary subterm, and a location with multiple outgoing edges corresponds to a local conflict between multiple terms.

These terms also carry information to support additional operations. To enable the reconstruction of the original graph, each constructor carries the UID of its original vertex, each reference carries the vertex that it refers to, and each subterm carries the UID of the edge that leads to it. To support type hole inference (as discussed in Section 5.2), each empty hole and local conflict carries its



(a) A graph

$$\Theta = \{(^{(40,*,L)}\square *^{40}\ (^{43}\curlyvee_{43,(42,x)}))\ +^{38}\ (^{41}\curlyvee_{41,(42,x)}),\ z^{52}, x^{42}, {}^{(46,+,L)}\square +^{46}\ (^{45}\circlearrowright_{45,(46,+)}\ *^{48}\ (^{49}y^{50}))\}$$

(b) The grove corresponding to the graph

$$\begin{aligned} t_r &= {}^{1}(^{(40,*,L)}\square *^{40}\ (^{43}\curlyvee_{43,(42,x)}))\ +^{38}\ (^{41}\curlyvee_{41,(42,x)}) \\ NP &= \{z^{52}\} \\ MP &= \{x^{42}\} \\ U &= \{^{(46,+,L)}\square +^{46}\ (^{45}\circlearrowright_{45,(46,+)}\ *^{48}\ (^{49}y^{50}))\} \end{aligned}$$

(c) The partitioned grove corresponding to the graph

Fig. 12. Example of graph decomposition

location. Finally, to support user navigation of the grove, each reference also carries the UID of the edge that leads to it.

3.2.2 Graph Decomposition. A live subgraph G, which is just a finite set of edges, decomposes into a grove Θ, which is a finite set of terms. In a tree, each node except the root has a unique parent, and there are no cycles. There is no guarantee of these properties in a graph. Decomposition operates by allowing each 'problematic' vertex, in the sense of having multiple parents or forming a cycle, to be the root of its own entry in the grove, and to use references at special term leaves to encode the original graph structure.

Formally, the parents of a vertex v in a graph is defined as the set of origin vertices v' such that the graph includes an edge from v' to v. Conversely, the children of a location ℓ is the set of v such that the graph includes an edge from ℓ to v. It is convenient to consider these children to be paired with its edge's UID .

Definition 3.3.

$$\begin{aligned} \mathsf{parents}_G(v) &= \{v' \in \mathcal{V} \mid (w, (v', p), v) \in G \wedge w \in \mathcal{W} \wedge p \in \mathcal{P}\} \\ \mathsf{children}_G(\ell) &= \{(w, v) \in \mathcal{W} \times \mathcal{V} \mid (w, \ell, v) \in G\} \end{aligned}$$

To help define decomposition, each vertex in a graph can be classified as either an NP root (if it has no parents), an MP root (if it has multiple parents), a U root (if there is a sequence of unique parents from it to itself, and it has the minimal UID in that sequence), or not a root. These classes are mutually exclusive.

Definition 3.4. A vertex v is an *NP root* in G if $\mathsf{parents}_G(v) = \varnothing$

Definition 3.5. A vertex v is an *MP root* in G if $\left|\mathsf{parents}_G(v)\right| > 1$

Definition 3.6. A vertex v is a *U root* in G if $\exists \{v_i\}_{1 \leqslant i \leqslant n}$ such that $v_1 = v_n = v$, $\mathsf{parents}_G(v_i) = \{v_{i+1}\}$ for all $1 \leqslant i < n$, and the UID of v is the minimum of the UIDs of $\{v_i\}_{1 \leqslant i \leqslant n}$.

Definition 3.7. $\mathsf{root}_G(v)$ if v is an NP root, an MP root, or a U root in G.

Lemma 3.8 (Unique Classification). *For any live subgraph $G \subseteq \mathcal{E}$ and vertex $v \in V$, exactly one of the following holds:*

- *v is an NP root in G*
- *v is an MP root in G*
- *v is an U root in G*
- *v is not a root in G*

The decomposition of a vertex is specified intuitively as the term obtained by traversing the descendants of the vertex until a leaf or a root vertex of some kind is reached, which is not further decomposed, but is denoted in the term as the appropriate reference. The decomposition of a graph is simply the set obtained by decomposing each vertex classified as a root, provided the vertex actually appears as the origin of an edge in G.

Formally, the decomposition of a vertex is given by the function $\mathsf{decompVertex}$, which returns a term with the same UID and constructor as the vertex. At each position in the arity of the constructor, we obtain a child term from the function $\mathsf{decompLoc}$, which decomposes a location in the graph to a corresponding child term. Consider the multiplication constructor with id 40 in Figure 12. It decomposes to a multiplication term with left and right child terms, the result of $\mathsf{decompLoc}$ applied to its left and right locations.

Definition 3.9. $\mathsf{decompVertex}_G(v = (u, k)) = k^u\ \{\mathsf{decompLoc}_G((v, p))\}_{p \in arity(k)}$

The decomposition $\mathsf{decompLoc}(\ell)$ proceeds by inspecting the children of location ℓ. If there are no children, the resulting child term is an empty hole. This hole is annotated with ℓ to act as a unique identifier. If ℓ has children, each is annotated with its edge UID and decomposed using the function $\mathsf{decompChild}$. If there are multiple children, they are placed in a local conflict, also annotated with ℓ. To continue our example, the left position of vertex 40 has no edges, and therefore the result is a hole annotated with the source $(40, *, L)$. The right position has one edge, so the right argument to the multiplication term is the result of applying $\mathsf{decompChild}$ to the destination vertex of this edge.

Definition 3.10.

$$\mathsf{decompLoc}_G(\ell) = \begin{cases} {}^{\ell}\square & \mathsf{children}_G(\ell) = \varnothing \\ {}^{w}\mathsf{decompChild}_G(w, v') & abs\mathsf{children}_G(\ell) = 1 \\ {}^{\ell}\{{}^{w}\mathsf{decompChild}_G(w, v') \mid (w, v') \in \mathsf{children}_G(\ell)\} & \left|\mathsf{children}_G(\ell)\right| > 1 \end{cases}$$

A call to $\mathsf{decompChild}$ is like a call to $\mathsf{decompVertex}$, except that roots are not decomposed further, and a reference is returned instead. Without this case, decomposition could traverse cycles in the graph and never terminate. Instead, decomposition results in well-behaved trees that nevertheless store enough information to enable rich exploration and analysis. In our example, vertex 42 has multiple parents, and therefore is an MP root. Its decomposition is a reference storing the edge UID 41 and the destination vertex.

Definition 3.11.

$$\mathsf{decompChild}_G(w, v) = \begin{cases} \curlyvee_{(w,v)} & v \text{ is an MP root in } G \\ \circlearrowright_{(w,v)} & v \text{ is a U root in} G \\ \mathsf{decompVertex}_G(v) & v \text{ is not a root in} G \end{cases}$$

Be decomposing each root vertex that appears in G, we obtain the whole grove.

Definition 3.12. $\mathsf{vertices}(G) = \left\{v \in \mathcal{V} \mid (w, (v, p), v') \in G \wedge w \in \mathcal{W} \wedge v' \in \mathcal{V} \wedge p \in \mathcal{P}\right\}$

Definition 3.13. $\mathsf{decomp}(G) = \left\{\mathsf{decompVertex}_G(v) \mid \mathsf{root}_G(v) \wedge v \in \mathsf{vertices}(G)\right\}$

The process can now be reversed and the same live subgraph can be obtained from the grove. At a high level, a term may be considered a tree, and therefore a graph, and the recomposition of a grove may be considered the simple union of each tree in the grove. Formally, the recomposition of a constructor applied to children is a set of edges that includes an edge from the parent to each child, as well as the recomposition of each child. Terms store vertices on constructors and edge UIDs for each child to enable the completeness of this reconstruction.

The recomposition of a constructor term is the set obtained by adding edges from each child of the term, according to the function recompChildTerm, whereas references contain no graph structure of their own and recompose to the empty set.

Definition 3.14.

$$\begin{aligned}
\mathsf{recompTerm}(k^u\ \{\bar{t}_p\}_{p\in arity(k)}) &= \bigcup_{p\in\mathsf{arity}(k)} \mathsf{recompChildTerm}(((u,k),p),\bar{t}_p)\\
\mathsf{recompTerm}(\curlyvee_{(w,v)}) &= \varnothing\\
\mathsf{recompTerm}(\circlearrowright_{(w,v)}) &= \varnothing
\end{aligned}$$

The recomposition of a child term at location ℓ includes, for each term in ℓ, an edge from the ℓ to the term's corresponding vertex, as well as the term's recomposition. In the case of an empty hole, there are no terms, and this the recomposition is empty. If there are one or more terms, each term contributes its own edge and recomposition.

Definition 3.15.

$$\begin{aligned}
\mathsf{recompChildTerm}(\ell, {}^{\ell}\square) &= \varnothing\\
\mathsf{recompChildTerm}(\ell, {}^{w}t) &= \left\{(w,\ell,\mathsf{vertexOfTerm}(t))\right\} \cup \mathsf{recompTerm}(t)\\
\mathsf{recompChildTerm}(\ell, {}^{\ell}\{{}^{w_i}t_i\}_{i<n}) &= \bigcup_{i<n}(\left\{(w_i,\ell,\mathsf{vertexOfTerm}(t_i))\right\} \cup \mathsf{recompTerm}(t_i))
\end{aligned}$$

The vertex of a term is used as the destination of edges from the term's parent location. Thus the vertex of a constructor term is the vertex with the same UID and constructor, and the vertex of a reference is the vertex it refers to.

Definition 3.16.

$$\begin{aligned}
\mathsf{vertexOfTerm}(k^u\ \{\bar{t}_p\}_{p\in arity(k)}) &= (u,k)\\
\mathsf{vertexOfTerm}(\curlyvee_{(w,v)}) &= v\\
\mathsf{vertexOfTerm}(\circlearrowright_{(w,v)}) &= v
\end{aligned}$$

The recomposition of a grove is simply the union of the recompositions of its terms.

Definition 3.17. $\mathsf{recomp}(T) = \bigcup_{t\in T} \mathsf{recompTerm}(t)$

The Theorem 3.18 states that recomposition recovers the entirety of the original live subgraph, and therefore that groves faithfully represent the underlying data structure.

Theorem 3.18 (Recomposability). *For all subgraphs G,* $\mathsf{recomp}(\mathsf{decomp}(G)) = G$.

$$
\begin{array}{lll}
\alpha \in \mathit{Act} & ::= & \mathsf{Construct}(k) \mid \mathsf{Delete} \mid \mathsf{Relocate}(\ell) \\
\hat{t} \in \mathit{ZTerm} & ::= & \rhd t \lhd \mid k^u\ \{\overline{t}\}_{p \in \mathit{arity}(k) \setminus \{\hat{p}\}}\ \hat{\overline{t}} \\
\hat{\overline{t}} \in \mathit{ZChildTerm} & ::= & \rhd^{\ell} \square \lhd \mid \rhd^{\ell} \{^{w_i} t_i\}_{i<n} \lhd \mid {}^{w}\hat{t} \mid {}^{\ell}\{t_i\}_{i<n, i \neq j}\ \hat{t}
\end{array}
$$

Fig. 13. The syntax of zippered terms and user actions

3.2.3 Partitioned Groves. In order to facilitate user interaction with groves, we defined a presentation of a grove called a *partitioned grove*. A partitioned grove contains three classes, corresponding to the the three kinds of roots: NP, MP, and U. A partitioned grove also designates a distinguished child term as the primary component of the program.

Formally, a partitioned grove γ is a quadruple $(\overline{t}_r, NP, MP, U)$ where $\overline{t}_r \in \mathit{ChildTerm}$ and NP, MP, U are finite sets of terms. The construction of a partitioned grove from the grove $\mathsf{decomp}(G)$ requires the designation of a *distinguished root* location $\ell_r = (v_r = (u_r, k_r), p_r)$ such that v_r is an NP root in G and $\mathsf{arity}(k_r) = \{p_r\}$. The distinguished child term can be thought of as the decomposition of the distinguished root location. The NP, MP, and U partitions are delineated by the class of the root vertex of each term. The root vertex of a term is like the vertex of a term, except that the root vertex of a reference is the *source* of the corresponding edge, rather than the destination. A term rooted at the distinguished root is excluded from the NP class, since its contents are already identified as the distinguished child term. Figure 12c provides an example of a partitioned grove, where the position Root of vertex 0 is the distinguished root location.

Definition 3.19.

$$
\begin{array}{lll}
\mathsf{distinguishedChildTerm}(\ell_r = ((u_r, k_r), p_r), \Theta) = \overline{t}_r & & \text{if } k_r^{u_r}\ \overline{t}_r \in \Theta \\
\mathsf{distinguishedChildTerm}(\ell_r = ((u_r, k_r), p_r), \Theta) = {}^{\ell_r}\square & & \text{otherwise}
\end{array}
$$

Definition 3.20.

$$
\begin{aligned}
\mathsf{rootVertexOfTerm}_G(k^u\ \{\overline{t}_p\}_{p \in \mathit{arity}(k)}) &= (u, k) \\
\mathsf{rootVertexOfTerm}_G(\curlyvee_{(w,v)}) &= v' \mathsf{where} (w, (v', p), v) \in G \\
\mathsf{rootVertexOfTerm}_G(\circlearrowright_{(w,v)}) &= v' \mathsf{where} (w, (v', p), v) \in G
\end{aligned}
$$

Definition 3.21.

$$
\begin{aligned}
\mathsf{grove}(\ell_r = (v_r, p_r), \Theta, G) = (&\mathsf{distinguishedChildTerm}(\ell_r, \Theta), \\
&\left\{t \in \Theta \mid \mathsf{rootVertexOfTerm}_G(t) \neq v_r \text{ and is an NP root in } G\right\}, \\
&\left\{t \in \Theta \mid \mathsf{rootVertexOfTerm}_G(t) \text{ is an MP root in } G\right\}, \\
&\left\{t \in \Theta \mid \mathsf{rootVertexOfTerm}_G(t) \text{ is a U root in } G\right\})
\end{aligned}
$$

3.3 User Edit Actions

We have defined a patch language directly in terms of graph edges and edge states. Since the user interacts with the more intuitive, tree-based grove representation of the program, we require an equally intuitive system of *user edit actions* (edits) that can be translated into the patch language. The new sorts are defined in Figure 13.

Edit actions operate on zipper terms [13, 25]. The definitions of zipper terms and child terms are standard, except that the cursor cannot select a child term of the form ${}^{w}t$, but is forced to select the

term t directly. While inspecting a grove, the user may select an edit action α while the cursor is represented by $\hat{t}$ or $\hat{\bar{t}}$, and this will result in a graph patch $\bar{\pi}$ to be applied directly to the graph.

$\mathsf{Construct}(k)$ constructs a new edge whose destination is determined by the cursor's location. If the cursor is on an empty hole, the hole is filled by constructing a fresh vertex with constructor k. If the cursor is on an existing term or child term, we *wrap* the existing term or child term by deleting it from its parent, constructing the new term in its place, then re-connecting the original term as a child of the new term.

Delete deletes any edges that pass from constructors above the cursor to constructors below the cursor. If an empty hole is selected, deletion is a no-op. If a local conflict is selected, each edge from that location to a conflicting child is deleted. If a constructor is selected, every edge targeting that constructor is deleted. If a reference is selected, the corresponding edge is deleted.

$\mathsf{Relocate}(\ell)$ combines edge construction and deletion into a single *atomic* operation. Provided the location ℓ is empty, the edges through the cursor are deleted, as described above, and new edges are simultaneously constructed with the same destinations as the old, but originating at location ℓ. This is not equivalent to the composition of a sequence of deletions and constructions, because the UIDs of the relocated vertices remain the same.

Formally, these patches are defined in terms of the function edges_G (Definition 3.22) which produces the set of edges in the live subgraph G passing through a cursor at the given term or child term. The live subgraph G is therefore an additional input to the patch construction judgment.

Definition 3.22.

$$
\begin{array}{rcl}
\mathsf{edges}_G({}^{\ell}\square) & = & \varnothing \\
\mathsf{edges}_G({}^{\ell}\{{}^{w_i} t_i\}_{i<n}) & = & \{(w_i, \ell, v) \mid (w_i, \ell, v) \in G \wedge i < n \wedge \ell \in \mathcal{L} \wedge v \in \mathcal{V}\} \\
\mathsf{edges}_G(k^{u}\ \{\bar{t}_p\}_{p \in arity(k)}) & = & \{(w, \ell, (u, k)) \mid (w, \ell, (u, k)) \in G \wedge w \in \mathcal{W} \wedge \ell \in \mathcal{L}\} \\
\mathsf{edges}_G(\curlyvee_{(w,v)}) & = & \{(w, \ell, v) \mid (w, \ell, v) \in G \wedge \ell \in \mathcal{L}\} \\
\mathsf{edges}_G(\circlearrowright_{(w,v)}) & = & \{(w, \ell, v) \mid (w, \ell, v) \in G \wedge \ell \in \mathcal{L}\}
\end{array}
$$

For the purpose of the construction action, it is assumed that each constructor $k \in \mathcal{K}$ in the language is equipped with a designated position $\mathsf{defaultPos}(k) \in \mathsf{arity}(k)$. The formal patch generation judgment is defined in terms of edges and defaultPos in Figure 14.

3.4 Mechanized Metatheory

The definitions and theorems in Section 3.1 and Section 3.2.2 have been mechanized [1] in the Agda proof assistant, with the exception that termination is justified separately for some definitions and proofs. In particular, Theorem 3.2 and Theorem 3.18 are formalized and proven. Lemma 3.8 follows from the more powerful theorem of classification correctness and completeness, which is mechanized and used to prove Theorem 3.18.

4 The Grove Workbench

We implemented the core Grove calculus of the previous section as an `OCaml` library called the Grove Workbench [1] and a corresponding web-based collaborative structure editor written using `js_of_ocaml` [41] primarily intended to demonstrate the collaborative features of the workbench and serve as a companion to the formal developments in this paper. The library is parameterized by a syntax specification for expressions, with the necessary data structures generated automatically.

4.1 The Grove Workbench

On opening up the *Grove Workbench*, the user is met with two almost identical panels side-by-side, emulating a collaborative editor environment between two users. Additional collaborators can

$$\boxed{\hat{t} \xrightarrow{\alpha,G} \bar{\pi}} \quad \boxed{\hat{\bar{t}} \xrightarrow{\alpha,G} \bar{\pi}}$$

$$\frac{\hat{\bar{t}} \xrightarrow{\alpha,G} \bar{\pi}}{(k^u\ \{\bar{t}\}_{p \in arity(k) \setminus \{\hat{p}\}}\ \hat{\bar{t}}) \xrightarrow{\alpha,G} \bar{\pi}} \qquad \frac{\hat{t} \xrightarrow{\alpha,G} \bar{\pi}}{({}^w\hat{t}) \xrightarrow{\alpha,G} \bar{\pi}} \qquad \frac{\hat{t} \xrightarrow{\alpha,G} \bar{\pi}}{({}^\ell\{t_i\}_{i<n, i\neq j}\ \hat{t}) \xrightarrow{\alpha,G} \bar{\pi}}$$

DeleteTerm

$$\frac{}{\triangleright t \triangleleft \xrightarrow{\mathsf{Delete},G} \{(⊟, \varepsilon) \mid \varepsilon \in \mathsf{edges}_G(t)\}}$$

DeleteChildTerm

$$\frac{}{\triangleright \bar{t} \triangleleft \xrightarrow{\mathsf{Delete},G} \{(⊟, \varepsilon) \mid \varepsilon \in \mathsf{edges}_G(\bar{t})\}}$$

RelocateTerm

$$\frac{}{\triangleright t \triangleleft \xrightarrow{\mathsf{Relocate}(\ell),G} \{(⊟, (w, \ell', v)), (⊞, (w^+, \ell, v)) \mid (w, \ell', v) \in \mathsf{edges}_G(t)\}}$$

RelocateChildTerm

$$\frac{}{\triangleright \bar{t} \triangleleft \xrightarrow{\mathsf{Relocate}(\ell),G} \{(⊟, (w, \ell', v)), (⊞, (w^+, \ell, v)) \mid (w, \ell', v) \in \mathsf{edges}_G(\bar{t})\}}$$

ConstructTerm

$$\frac{\ell' = ((u^+, k), \mathsf{defaultPos}(k))}{\triangleright t \triangleleft \xrightarrow{\mathsf{Construct}(k),G} \{(⊟, (w, \ell, v)), (⊞, (w_1^+, \ell, v')), (⊞, (w_2^+, (v', \ell', v)) \mid (w, \ell, v) \in \mathsf{edges}_G(t)\}}$$

ConstructChildTerm

$$\frac{\bar{t} \neq {}^\ell\square \qquad \ell' = ((u^+, k), \mathsf{defaultPos}(k))}{\triangleright t \triangleleft \xrightarrow{\mathsf{Construct}(k),G} \{(⊟, (w, \ell, v)), (⊞, (w_1^+, \ell, v')), (⊞, (w_2^+, (v', \ell', v)) \mid (w, \ell, v) \in \mathsf{edges}_G(t)\}}$$

ConstructHole

$$\frac{}{\triangleright {}^\ell\square \triangleleft \xrightarrow{\mathsf{Construct}(k),G} \{(⊞, (w^+, \ell, (u^+, k)))\}}$$

Fig. 14. The patch generation judgment

be generated on command. In each case, a cursor is placed on an empty hole at the root of the displayed term decomposition, which is displayed as a graph visualization of the same graph structure so that the UI resembles the figures in Section 2. Below this are buttons for various user edit actions, and commands to send queued commands to specific other users. In addition, we have separate panels for multi-parented, deleted, and unicycle panels, corresponding to the partitioned grove datastructure in the previous section. We will now examine them in correspondence to the formalism described in Section 3.

4.2 Graph Implementation

The graph data structure is implemented as `OCaml Map` data structure with insertion and selection operations whose asymptotic worst-case complexity is logarithmic with respect to the size of the map. Since we cannot implement an infinite mapping directly, the graph only maps live or deleted edges to edge states $\{⊞, ⊟\}$. We do not represent edges that map to $\bot$.

For a graph $G : (\mathcal{E} = \mathcal{U} \times \mathcal{V} \times \mathcal{P} \times \mathcal{V}) \to \Sigma$, our graph decomposition algorithm runs in $O(|\mathcal{V}| \log |\mathcal{V}| + |\mathcal{E}| \log |\mathcal{V}|)$. It begins with a scan of all edges that have been created or deleted $O(|\mathcal{E}|)$. Their vertices at both ends are partitioned into three sets: multi-parented, single-parented, or

orphaned. Their relationships are recorded in maps for $O(\log |\mathcal{V}|)$ lookups of parent and child edge sets. After the vertices have been partitioned, we traverse the various single-parented components and produce equivalent expressions $O(|\mathcal{V}|)$. For unicycles, we traverse backward until a vertex is seen twice $O(|\mathcal{V}|)$, then proceed forward to find the least vertex on the cycle $O(|\mathcal{V}|)$. Once the least vertex is found, decomposition of unicycle begins with it, thus ensuring any edges to it become references to a unicycle root.

5 Total Type Error Localization and Recovery for Groves

Resolving conflicts is often a time-consuming process and requires understanding the semantics of the program even while it still has conflicts. Traditionally, conflicts are indicated by inserting extra-linguistic conflict markers into files.

These markers can limit the operation of language services that need a well-formed or well-typed term, e.g., type error reporting. Ignoring the markers is not sufficient because multiple conflicting versions of the code may appear between the markers. This can complicate variable resolution and type checking.

In the previous sections, we developed a system for explicitly representing various forms of syntactic conflict directly in the grove resulting from graph decomposition. We now consider the static semantics of groves.

Groves are sets of terms with holes, local conflicts, relocation conflict references, and unicycle conflict references between them. These terms may not yet be well-typed during the course of a collaboration. As such, we need to consider the problem of type error localization and recovery.

We build on recent work on the marked lambda calculus [43], which we briefly review in Section 5.1. The marked lambda calculus specifies a *total* type error localization system, i.e. one where every syntactically well-formed term can be *marked* with errors to produce a statically meaningful term.

In this section, we extend the marked lambda calculus to allow reasoning about conflicted programs, i.e. groves (instantiated with the syntax of the simply typed lambda calculus). The *marked grove calculus* maintains the key totality property, so *every edit state that can arise in the course of a collaboration is statically meaningful.* Downstream language services can, therefore, continue to provide support while users resolve conflicts. This addresses the **semantic gap problem** described in Section 1.

We follow the marked lambda calculus in basing our *marked grove calculus* in bidirectional type checking [8] and deploying gradual typing [36] when conflicts do not allow a single type to be inferred, then layer on a unification-based type inference system to opportunistically fill holes or suggest partial solutions when there are conflicting types due to conflicting syntax. We give an overview of the key judgments and rules in Section 5.2 but due to space considerations and because much of the development is based directly on the prior work, we leave the full details to the supplemental material and the mechanized metatheory, which is briefly outlined in 5.3.

5.1 Background: The Marked Lambda Calculus

The marked lambda calculus is a gradual bidirectionally typed rewriting system that takes an arbitrary syntactically well-formed expression and *marks* it by localizing type errors, producing a *marked expression.*

The key judgments are synthetic and analytic marking judgments that mark and type-check expressions: $\Gamma \vdash e \rightsquigarrow \check{e} \Rightarrow \tau$ and $\Gamma \vdash e \rightsquigarrow \check{e} \Leftarrow \tau$. The synthetic judgment is used when a type is to be locally inferred from e, while the analytic judgment is used when surrounding type annotations determine an expected type for e. A subsumption rule, which is not shown, allows analysis at any type consistent with the synthesized type.

For example, the rules for marking variables are reproduced below:

$$\textsc{MKSVar}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \hookrightarrow x \Rightarrow \tau} \qquad \textsc{MKSFree}\ \frac{x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash x \hookrightarrow (\!|x|\!)_{\square} \Rightarrow ?}$$

The first rule handles bound variables. The second rule handles free variables, which would normally be statically meaningless. Here, we instead mark it explicitly as a free variable and synthesize the unknown type, ?.

The rules for marking lambda abstractions when an expected type is provided similarly handle both well-typed and ill-typed cases (which is key to proving totality):

$$\textsc{MKALam1}\ \frac{\tau_3 \blacktriangleright_{\rightarrow} \tau_1 \rightarrow \tau_2 \qquad \tau \sim \tau_1 \qquad \Gamma, x : \tau \vdash e \hookrightarrow \check{e} \Leftarrow \tau_2}{\Gamma \vdash \lambda x : \tau . e \hookrightarrow \lambda x : \tau . \check{e} \Leftarrow \tau_3}$$

$$\textsc{MKALam2}\ \frac{\tau_3 \blacktriangleright_{\rightarrow} \qquad \Gamma, x : \tau \vdash e \hookrightarrow \check{e} \Leftarrow ?}{\Gamma \vdash \lambda x : \tau . e \hookrightarrow (\!|\lambda x : \tau . \check{e}|\!)^{\Leftarrow}_{\blacktriangleright_{\rightarrow}} \Leftarrow \tau_3} \qquad \textsc{MKALam3}\ \frac{\tau_3 \blacktriangleright_{\rightarrow} \tau_1 \rightarrow \tau_2 \qquad \tau \not\sim \tau_1 \qquad \Gamma, x : \tau_1 \vdash e \hookrightarrow \check{e} \Leftarrow \tau_2}{\Gamma \vdash \lambda x : \tau . e \hookrightarrow (\!|\lambda x : \tau . \check{e}|\!)_{:} \Leftarrow \tau_3}$$

The first rule invokes the matched arrow judgment (which handles the situation where the expected type is unknown [36]) and performs a type consistency check with the argument type annotation, after which we can analyze the body against the expected output type while recursively marking it.

If there is not a matched arrow type, the second rule marks the lambda with an error (while recovering recursively into the body at an unknown type). If the argument type annotation is inconsistent, the third rule similarly marks the lambda with a different error. The subscript and superscript are formal analogs of type error messages.

Marking is total, meaning every well-formed expression can be marked in this way, producing a well-typed marked expression as governed by the bidirectional typing judgments for marked expressions shown here: $\Gamma \vdash_{M} \check{e} \Rightarrow \tau$, $\Gamma \vdash_{M} \check{e} \Leftarrow \tau$.

These typing rules emit constraints because once marked, the system layers on unification-based type inference to opportunistically fill the unknown types, i.e., the type holes, that arise. In case the constraints governing a type hole cannot be solved, the system suggests hole fillings that satisfy a subset of the constraints, asking the user to resolve the issue (rather than attempting to heuristically decide which constraints were intended). When the user hovers over a choice, the system returns control to the bidirectional marking system.

5.2 Marking Groves

We extend the marked lambda calculus to groves. The machinery related to type errors remains largely unchanged. We focus our attention on type error localization in the presence of conflicts.

5.2.1 Syntax. Figure 15 introduces the syntax of the marked grove calculus. We define an *unmarked* language, which comprises unmarked expressions, e, child expressions, $\bar{e}$, types, τ, and child types $\bar{\tau}$. We also define a corresponding *marked* language, which comprises marked expressions, $\check{e}$, child expressions, $\check{\bar{e}}$, types, , and child types, . This syntax is an instantiation of the generic term syntax introduced in Section 3.2. The marked language differs from the unmarked language only by the inclusion of the marks that arise due to type errors taken directly from the prior work.

$$
\begin{array}{llll}
\tau & \in & \text{UType} & ::= \ \text{num}^u \mid \tau_1 \to^u \tau_2 \mid \curlyvee_{(w,v)} \mid \circlearrowright_{(w,v)} \\
\bar{\tau} & \in & \text{UChildType} & ::= \ {}^{\ell}\square \mid {}^{w}\tau \mid {}^{\ell}\{{}^{w_i}\tau_i\}_{i<n} \\
e & \in & \text{UExp} & ::= \ x^u \mid \underline{n}^u \mid \bar{e}_1 +^u \bar{e}_2 \mid \bar{e}_1 *^u \bar{e}_2 \mid \lambda^u x : \bar{\tau}.\bar{e} \mid (\bar{e}_1\ \bar{e}_2)^u \mid \curlyvee_{(w,v)} \mid \circlearrowright_{(w,v)} \\
\bar{e} & \in & \text{UChildExp} & ::= \ {}^{\ell}\square \mid {}^{w}e \mid {}^{\ell}\{{}^{w_i}e_i\}_{i<n} \\
\\
\check{e} & \in & \text{MExp} & ::= \ x^u \mid \underline{n}^u \mid \check{\bar{e}}_1 +^u \check{\bar{e}}_2 \mid \check{\bar{e}}_1 *^u \check{\bar{e}}_2 \mid \lambda^u x : \bar{\tau}.\bar{e} \mid (\check{\bar{e}}_1\ \check{\bar{e}}_2)^u \mid \curlyvee_{(w,v)} \mid \circlearrowright_{(w,v)} \\
& & & \quad (\!|x^u|\!)_{\square} \mid (\!|\lambda x^u : \bar{\tau}.\ \check{\bar{e}}|\!)_{:} \mid (\!|\lambda x^u : \bar{\tau}.\ \check{\bar{e}}|\!)^{\Leftarrow}_{\blacktriangleright\to} \mid ((\!|\check{\bar{e}}_1|\!)^{\Rightarrow}_{\blacktriangleright\to}\ \check{\bar{e}}_2)^u \mid (\!|\check{\bar{e}}|\!)_{\not\sim} \\
\check{\bar{e}} & \in & \text{MChildExp} & ::= \ {}^{\ell}\square \mid {}^{w}\check{e} \mid {}^{\ell}\{{}^{w_i}\check{e}_i\}_{i<n} \\
\\
\sigma & \in & \text{Type} & ::= \ ? \mid \text{num} \mid \sigma_1 \to \sigma_2 \\
\\
\dot{\sigma} & \in & \text{ProvType} & ::= \ ?^q \mid \text{num} \mid \dot{\sigma}_1 \to \dot{\sigma}_2 \\
q & \in & \text{Provenance} & ::= \ \text{typ}(\ell) \mid \text{exp}(\ell) \mid \text{ref}(w) \mid \text{mark}(u) \mid \to_L (q) \mid \to_R (q) \mid \text{anon} \\
m & \in & \text{Mode} & ::= \ \text{syn} \mid \text{ana}(\sigma)
\end{array}
$$

Fig. 15. Syntax

5.2.2 Graph Erasure. The presence of conflicts and UIDs requires us to distinguish syntactic types from semantic types, σ, which are related by a graph erasure function, $\tau^{\triangle} = \sigma$ and $\bar{\tau}^{\triangle} = \sigma$, that replaces holes and conflicts with the unknown type and erases vertex and edge UIDs (not shown, see supplement for the full definition).

5.2.3 Marking. Marking is performed bidirectionally using four mutually defined judgments: $\Gamma \vdash e \hookrightarrow \check{e} \Rightarrow \sigma$, $\Gamma \vdash \bar{e} \hookrightarrow \check{\bar{e}} \Rightarrow \sigma$, $\Gamma \vdash e \hookrightarrow \check{e} \Leftarrow \sigma$, and $\Gamma \vdash \bar{e} \hookrightarrow \check{\bar{e}} \Rightarrow \sigma$.

For standard forms in the simply typed lambda calculus, the rules correspond directly to those from the marked lambda calculus. For the sake of brevity, we include only the rules related to lambda expressions below; the full set of rules are in the supplemental material.

MKALam1
$$\frac{\bar{\tau}^{\triangle} = \sigma \qquad \sigma_3 \blacktriangleright_{\to} \sigma_1 \to \sigma_2 \qquad \sigma \sim \sigma_1 \qquad \Gamma,\ x : \sigma \vdash \bar{e} \hookrightarrow \check{\bar{e}} \Leftarrow \sigma_2}{\Gamma \vdash \lambda^u x : \bar{\tau}.e \hookrightarrow \lambda^u x : \bar{\tau}.\check{\bar{e}} \Leftarrow \sigma_3}$$

MKALam2
$$\frac{\bar{\tau}^{\triangle} = \sigma \qquad \sigma_3 \blacktriangleright_{\to} \qquad \Gamma,\ x : \sigma \vdash \bar{e} \hookrightarrow \check{\bar{e}} \Leftarrow ?}{\Gamma \vdash \lambda^u x : \bar{\tau}.\bar{e} \hookrightarrow (\!|\lambda x^u : \bar{\tau}.\ \check{\bar{e}}|\!)^{\Leftarrow}_{\blacktriangleright\to} \Leftarrow \sigma_3}$$

MKALam3
$$\frac{\bar{\tau}^{\diamond} = \sigma \qquad \sigma_3 \blacktriangleright_{\to} \sigma_1 \to \sigma_2 \qquad \sigma \not\sim \sigma_1 \qquad \Gamma,\ x : \sigma_1 \vdash \bar{e} \hookrightarrow \check{\bar{e}} \Leftarrow \sigma_2}{\Gamma \vdash \lambda^u x : \bar{\tau}.\bar{e} \hookrightarrow (\!|\lambda x^u : \bar{\tau}.\ \check{\bar{e}}|\!)_{:} \Leftarrow \sigma_3}$$

These rules differ from the original rules, reproduced in Section 5.1, only in that they graph-erase the type annotation and recurse using the marking rules for child expressions. For empty holes

and singleton expressions, these are straightforward. We return to local conflicts below.

$$\textsc{MKSHole}\ \frac{}{\Gamma \vdash {}^{\ell}\square \hookrightarrow {}^{\ell}\square \Rightarrow ?} \qquad \textsc{MKSOnly}\ \frac{\Gamma \vdash e \hookrightarrow \check{e} \Rightarrow \sigma}{\Gamma \vdash {}^{w}e \hookrightarrow {}^{w}\check{e} \Rightarrow \sigma} \qquad \textsc{MKAHole}\ \frac{}{\Gamma \vdash {}^{\ell}\square \hookrightarrow {}^{\ell}\square \Leftarrow \sigma} \qquad \textsc{MKAOnly}\ \frac{\Gamma \vdash e \hookrightarrow \check{e} \Leftarrow \sigma}{\Gamma \vdash {}^{w}e \hookrightarrow {}^{w}\check{e} \Leftarrow \sigma}$$

Once marked, the corresponding marked expression typing judgments, $\Gamma \vdash \check{e} \Rightarrow \dot{\sigma} \mid C \mid L$, $\Gamma \vdash \check{\check{e}} \Rightarrow \dot{\sigma} \mid C \mid L$, $\Gamma \vdash \check{e} \Leftarrow \dot{\sigma} \mid C \mid L$, and $\Gamma \vdash \check{\check{e}} \Leftarrow \dot{\sigma} \mid C \mid L$ emit type inference constraints, C, as in prior work, and, recursively collect location conflict contexts, L, discussed further below. When the system generates inference constraints and tries to unify them, it encounters unknown types that need to be distinguished. This motivates the need for $\dot{\sigma}$ in Figure 15, which extends the sort of semantic types to enable linking generated unknown types to their associated provenance denoted by q. Provenances help locate the origin of the unknown type by associating them with their location (ℓ), edge-ID (w), or vertex-ID (u).

$$\textsc{MALam1}\ \frac{\dot{\sigma}_3 \blacktriangleright_{\to} \dot{\sigma}_1 \to \dot{\sigma}_2 \mid C_1 \qquad \bar{\tau}^{\triangle} = \dot{\sigma} \qquad \dot{\sigma} \sim \dot{\sigma}_1 \qquad \Gamma,\ x : \dot{\sigma} \vdash \check{\check{e}} \Leftarrow \dot{\sigma}_2 \mid C_2 \mid L_1}{\Gamma \vdash \lambda^{u} x : \bar{\tau}.\check{\check{e}} \Leftarrow \dot{\sigma}_3 \mid C_1 \cup C_2 \cup \{\dot{\sigma} \approx \dot{\sigma}_1\} \mid L_1}$$

$$\textsc{MALam2}\ \frac{\dot{\sigma}_3 \blacktriangleright_{\to} \qquad \bar{\tau}^{\triangle} = \dot{\sigma} \qquad \Gamma,\ x : \dot{\sigma} \vdash \check{\check{e}} \Leftarrow ?^{\mathsf{anon}} \mid C \mid L}{\Gamma \vdash (\!|\lambda x^{u} : \bar{\tau}.\ \check{\check{e}}|\!)^{\Leftarrow}_{\blacktriangleright_{\to}} \Leftarrow \dot{\sigma}_3 \mid C \cup \left\{?^{\mathsf{mark}(u)} \approx \dot{\sigma}_3\right\} \mid L}$$

$$\textsc{MALam3}\ \frac{\dot{\sigma}_3 \blacktriangleright_{\to} \dot{\sigma}_1 \to \dot{\sigma}_2 \mid C_1 \qquad \bar{\tau}^{\triangle} = \dot{\sigma} \qquad \dot{\sigma} \not\sim \dot{\sigma}_1 \qquad \Gamma,\ x : \dot{\sigma}_1 \vdash \check{\check{e}} \Leftarrow \dot{\sigma}_2 \mid C_2 \mid L}{\Gamma \vdash (\!|\lambda x^{u} : \bar{\tau}.\ \check{\check{e}}|\!)_{:} \Leftarrow \dot{\sigma}_3 \mid C_1 \cup C_2 \cup \left\{?^{\mathsf{mark}(u)} \approx \dot{\sigma}_3\right\} \mid L}$$

5.2.4 Local Conflicts. To mark local conflicts, we recursively mark all conflicted terms under the given context and with the same expected type when available. The local conflict as a whole synthesizes the unknown type, i.e. it is essentially a "conflict hole":

$$\textsc{MKSLocalConflict}\ \frac{\left\{\Gamma \vdash e \hookrightarrow \check{e} \Rightarrow \sigma_i\right\}_{i<n}}{\Gamma \vdash {}^{\ell}\{{}^{w_i}e_i\}_{i<n} \hookrightarrow {}^{\ell}\{{}^{w_i}\check{e}_i\}_{i<n} \Rightarrow ?} \qquad \textsc{MKALocalConflict}\ \frac{\left\{\Gamma \vdash e_i \hookrightarrow \check{e}_i \Leftarrow \sigma\right\}_{i<n}}{\Gamma \vdash {}^{\ell}\{{}^{w_i}e_i\}_{i<n} \hookrightarrow {}^{\ell}\{{}^{w_i}\check{e}_i\}_{i<n} \Leftarrow \sigma}$$

After marking, we generate type inference constraints that constrain this hole (identified using a provenance based on the location of conflict, ℓ) using all of the conflicted terms.

$$\textsc{MSLocalConflict}\ \frac{\left\{\Gamma \vdash \check{e} \Rightarrow \dot{\sigma}_i \mid C_i \mid L_i\right\}_{i<n}}{\Gamma \vdash {}^{\ell}\{{}^{w_i}\check{e}_i\}_{i<n} \Rightarrow ?^{\mathsf{exp}(\ell)} \mid \bigcup_{i<n} C_i \cup \left\{?^{\mathsf{exp}(\ell)} \approx \dot{\sigma}_i\right\}_{i<n} \mid \bigcup_{i<n} L_i}$$

$$\textsc{MALocalConflict}\ \frac{\left\{\Gamma \vdash \check{e}_i \Leftarrow \dot{\sigma} \mid C_i \mid L_i\right\}_{i<n}}{\Gamma \vdash {}^{\ell}\{{}^{w_i}\check{e}_i\}_{i<n} \Leftarrow \dot{\sigma} \mid \bigcup_{i<n} C_i \mid \bigcup_{i<n} L_i}$$

When the conflicted terms have a consistent type, inference will succeed. When it fails, we can fall back to the unknown type and rely on the type hole inference system developed in prior work to allow users to choose from partial hole solutions. When a partial solution is selected (effectively annotating the conflict), the mode becomes analytic, and the conflicted terms are remarked, allowing users to identify where errors would arise if the conflict were resolved at a particular type.

5.2.5 Relocation and Unicycle Conflict References. Relocation and unicycle conflict references also synthesize the unknown type and operate similarly:

$$\textsc{MKSRelocationConflict}\quad \frac{}{\Gamma \vdash \curlyvee_{(w,v)} \rightsquigarrow \curlyvee_{(w,v)} \Rightarrow ?}$$

$$\textsc{MSRelocationConflict}\quad \frac{}{\Gamma \vdash \curlyvee_{(w,v)} \Rightarrow ?^{\mathsf{ref}(w)} \mid \{\} \mid (v, w, \Gamma, \mathsf{syn})}$$

$$\textsc{MARelocationConflict}\quad \frac{}{\Gamma \vdash \curlyvee_{(w,v)} \Leftarrow \dot{\sigma} \mid \left\{?^{\mathsf{ref}(v)} \approx \dot{\sigma}\right\} \mid (v, w, \Gamma, \mathsf{ana}(\dot{\sigma}))}$$

$$\textsc{MKSCycleLocationConflict}\quad \frac{}{\Gamma \vdash \circlearrowright_{(w,v)} \rightsquigarrow \circlearrowright_{(w,v)} \Rightarrow ?}$$

$$\textsc{MSUnicycleConflict}\quad \frac{}{\Gamma \vdash \circlearrowright_{(w,v)} \Rightarrow ?^{\mathsf{ref}(w)} \mid \{\} \mid (v, w, \Gamma, \mathsf{syn})}$$

$$\textsc{MAUnicycleConflict}\quad \frac{}{\Gamma \vdash \circlearrowright_{(w,v)} \Leftarrow \dot{\sigma} \mid \left\{?^{\mathsf{ref}(v)} \approx \dot{\sigma}\right\} \mid (v, w, \Gamma, \mathsf{ana}(\dot{\sigma}))}$$

Because the referenced term has multiple possible locations, it also has multiple possible typing contexts and typing modes, so we cannot immediately mark it. Instead, we gather this information in the location conflict context, L, as shown above. This maps a location (identified by an edge UID, w, and also for the vertex, v, for operational simplicity) to a pair of a typing context, Γ, and a typing mode, m. This information can be used by the system to provisionally mark the conflicted term under an explicitly selected context and mode (e.g. in response to which location the user's cursor is on or using some other user interface affordance).

The emitted constraints assume an unknown type for the referenced vertex and constrain it with any expected types that appear at any of the locations where the corresponding term appears. Again, solutions to this unknown type can be presented to the user to help them decide which location might be most sensible.

5.3 Mechanized Metatheory

The key meta-theoretic property that tells us that we did not miss any cases is *totality*:

Theorem 5.1 (Marking Totality).

(1) For all Γ and e, there exist $\check{e}$ and σ such that $\Gamma \vdash e \rightsquigarrow \check{e} \Rightarrow \sigma$ and $\Gamma \vdash \check{e} \Rightarrow \dot{\sigma} \mid C \mid L$ and $\dot{\sigma}^\circ = \sigma$.

(2) For all Γ, e, and σ, there exists $\check{e}$ such that $\Gamma \vdash e \rightsquigarrow \check{e} \Leftarrow \sigma$ and $\Gamma \vdash \check{e} \Leftarrow \dot{\sigma} \mid C \mid L$ and $\dot{\sigma}^\circ = \sigma$.

In addition, the prior work defined a number of other auxiliary metatheorems that help sanity check these definitions: well-formedness (marking preserves syntactic structure), and marking unicity (marking is deterministic).

We have mechanized [1] our extension of the marked lambda calculus and these metatheorems using the Agda proof assistant, taking the standard approach of modeling judgments as inductive

datatypes and inference rules as constructors. Marked terms are intrinsically typed, allowing L to be computed as a function of the term.

6 Related Work

The core components of a version control system are a patch language, a method for synthesizing patches from user actions, and an approach for merging patches.

6.1 Patch Languages

There have been many different designs for patch languages and indeed many imperative data structures and their associated operations can be construed as patch languages in the most general sense. In the context of collaborative coding, patch languages can differ in the data structures they operate on (e.g. line-based text [32], character-based text [9, 20], tree-structured data [16]). They can also differ in how they identify locations within the data (e.g. by using numeric offsets [9], unique identifiers, or paths through a tree). Finally, patch languages differ in which specific actions are supported explicitly. Insertion and deletion are common, while code relocation, copying, undo, and other operations are variously also included.

In this paper, our focus was on syntax trees with holes (i.e. program sketches [25, 38]) and explicit conflicts, which we represented as directed graphs. We identify locations using unique IDs. We have a two-level patch language, with a low-level graph patch language supporting only edge insertion and deletion and a higher-level user edit action language focused on insertion, deletion, and relocation, with some additional narrative consideration of copying and undo (a fuller account of which we leave to future work). Our user edit language therefore forms a structure editor calculus, inspired closely by recent work on the Hazelnut structure editor calculus (which did not support relocation) [25] and patch languages for other tree-based data structures [6, 10, 11, 19, 23, 34].

6.2 Patch Synthesis

The most common approach to patch synthesis is to use a differencing algorithm to compare two states, e.g. from the file system, to generate a patch. The classic `diff` algorithm [14], for example, minimizes edit distance for a patch language involving line insertions and deletions.

In the setting of tree-based editing, there have been a number of tree differencing algorithms described in the literature [4–7, 10, 11, 16, 19, 23, 34]. As described in Section 1, synthesizing insertions and deletions is well-understood, but synthesizing relocations is more complex and requires heuristics.

The approach we explore in this paper is far simpler: we directly translate from the log of user edit actions to graph patches, without needing a differencing algorithm at all. This is only possible with a structure editor integrated with the version control system, but the benefit of direct visibility into the edit action log is that we do not need heuristics to synthesize relocations.

6.3 Merging

6.3.1 Operational Transforms. The most common approach to merging concurrently developed patches is to deploy an operational transform [9] whereby locations in a remote patch are modified based on the action of a local patch. Standard three-way merge algorithms in version control systems deploy this approach, as do real-time collaborative rich text editors [15].

There have been a number of papers studying the algebraic properties of merging patches in this style. For example, the Darcs [32] version control system, like Grove, represents repositories using sets of patches. Using operational transforms, Darcs achieves commutativity in many cases, but not between conflicting patches. The theory of Darcs defines and algebraically characterizes when operational transforms do satisfy the properties of associativity and commutativity. Recent

work on homotopical patch theory [3] has similarly developed an abstract algebraic framework for distinguishing sensible merges.

6.3.2 CRDTs. The observation that commutativity is a particularly helpful property when dealing with concurrent systems, including version control systems, has led to the development of a number of data structures centered on commutativity. These are known as CRDTs, which stands variously for *conflict-free*, *convergent* (CvRDT), or *commutative replicated datatypes* (CmRDT) depending on particular details about the operations and the state representation [35]. Our approach draws directly from this line of work: the Grove patch language forms a CmRDT on directed graphs, from which we observe that we can derive a CmRDT for trees with explicit conflicts.

Much of the prior work on CRDT-based collaborative editing has focused on text editing [2, 12, 17, 22, 24, 27, 30, 31, 42], typically with text tagged with unique IDs. These techniques have been used to develop collaborative text editors, e.g. Zed [37].

Peritext develops a CRDT-based approach for collaboratively editing rich text, which is structured as text with hierarchically marked regions [20].

There have been other recent efforts to develop CmRDTs for richer tree data structures [18, 30]. However, the focus in this work has been on avoiding conflicts and cycles entirely by applying *ad hoc* heuristics for conflict resolution at merge-time, e.g. using reported timestamps or favoring particular directions in the tree. Our approach instead embraces manual conflict resolution, as is common practice in software projects where arbitrarily losing code is not acceptable.

Pijul iterates on the patch based system of Darcs, obtaining commutativity between conflicting patches using a CRDT graph data structure [44]. Pijul's graph data structure is very similar to that of Grove in its treatment of edges, but, unlike Grove, requires vertices to be created before they can be referred to. This imposes a dependency relation between patches, causing Pijul to fall short of the full commutativity enjoyed by Grove. Pijul is also language agnostic and models only the linear structure of text. However, Pijul goes beyond this version of Grove by extending the data structure to support history and branches, and Grove may be similarly extended in future work.

7 Discussion and Conclusion

> *"The fact that commutation can fail [in Darcs] makes a huge difference in the whole patch formalism. It may be possible to create a formalism in which commutation always succeeds, with the result of what would otherwise be a commutation that fails being something like a virtual particle ... and it may be that such a formalism would allow strict mathematical proofs ... However, I'm not sure how you'd deal with a request to delete a file that has not yet been created, for example. Obviously you'd need to create some kind of antifile, which would annihilate with the file when that file finally got created ..."*
> – David Roundy, Theory of patches [33]

This paper proposes a radically simpler, albeit practically ambitious, rearchitecture of collaborative editing. Our contributions together result in a typed collaborative structure calculus called Grove where, uniquely, all edits, including code relocations that stymie existing approaches, commute and where there are no semantic gaps: all possible editor states, including editor states with various kinds of unresolved conflicts, are semantically meaningful.

This paper focuses on the core theoretical underpinnings of this approach, developing mechanized metatheory for both the patch language and the type system. A number of research problems on algorithmic, networking, and user interface aspects of the problem open up given these foundations. For example, we point out several situations where presenting non-conflicted but heuristically attention-worthy merges may be worthwhile, and we leave to future work the user experience design of this process. We intend to use the Grove workbench to integrate these efforts into the

Hazel programming environment, though its editor component has been evolving so rapidly as to prevent experimentation in this direction so far.

Although our focus was on tree editing, some aspects of a program are more naturally linearly structured, e.g. string literals. We also leave to future work the problem of combining existing work on sequence CRDTs with our work on tree/graph CRDTs.

Data Availability Statement

The artifact [1] that accompanies this work contains the Agda mechanization of the theorems discussed in section 5.3 and section 3.4 and contains the implementation of the Grove Workbench discussed in section 4.

Acknowledgments

This work was partially funded by the National Science Foundation under Grant No. 2238744. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Michael Adams, Eric Griffis, Thomas Porter, Sundara Vishnu Satish, Eric Zhao, and Cyrus Omar. 2024. *Artifact for Grove: A Bidirectionally Typed Collaborative Structure Editor Calculus.* https://doi.org/10.5281/zenodo.14026532

[2] Mehdi Ahmed-Nacer, Claudia-Lavinia Ignat, Gérald Oster, Hyun-Gul Roh, and Pascal Urso. 2011. Evaluating crdts for real-time document editing. In *Proceedings of the 11th ACM symposium on Document engineering*. 103–112.

[3] Carlo Angiuli, Edward Morehouse, Daniel R. Licata, and Robert Harper. 2016. Homotopical patch theory. *J. Funct. Program.* 26 (2016), e18. https://doi.org/10.1017/S0956796816000198

[4] Taku Aratsu, Kouichi Hirata, and Tetsuji Kuboyama. 2010. Approximating Tree Edit Distance through String Edit Distance for Binary Tree Codes. *Fundam. Informaticae* 101, 3 (2010), 157–171. https://doi.org/10.3233/FI-2010-282

[5] Philip Bille. 2005. A survey on tree edit distance and related problems. *Theor. Comput. Sci.* 337, 1-3 (2005), 217–239. https://doi.org/10.1016/J.TCS.2004.12.030

[6] Sudarshan S. Chawathe and Hector Garcia-Molina. 1997. Meaningful Change Detection in Structured Data. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, Joan Peckham (Ed.). ACM Press, 26–37. https://doi.org/10.1145/253260.253266

[7] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. 2009. An optimal decomposition algorithm for tree edit distance. *ACM Trans. Algorithms* 6, 1 (2009), 2:1–2:19. https://doi.org/10.1145/1644015.1644017

[8] Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (May 2021), 38 pages. https://doi.org/10.1145/3450952

[9] Clarence A. Ellis and Simon J. Gibbs. 1989. Concurrency Control in Groupware Systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*, James Clifford, Bruce G. Lindsay, and David Maier (Eds.). ACM Press, 399–407. https://doi.org/10.1145/67544.66963

[10] 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

[11] 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

[12] Victor S. Grishchenko. 2010. Deep hypertext with embedded revision control implemented in regular expressions. In *Proceedings of the 6th International Symposium on Wikis and Open Collaboration, 2010, Gdansk, Poland, July 7-9, 2010*, Phoebe Ayers and Felipe Ortega (Eds.). ACM. https://doi.org/10.1145/1832772.1832777

[13] Gérard P. Huet. 1997. The Zipper. *J. Funct. Program.* 7, 5 (1997), 549–554. https://doi.org/10.1017/S0956796897002864

[14] James W. Hunt and M. Douglas McIlroy. 1976. An Algorithm for Differential File Comparison. (1976). https://www.cs.dartmouth.edu/%7Edoug/diff.pdf

[15] Claudia-Lavinia Ignat, Luc André, and Gérald Oster. 2021. Enhancing rich content wikis with real-time collaboration. *Concurr. Comput. Pract. Exp.* 33, 8 (2021). https://doi.org/10.1002/CPE.4110

[16] Philip N. Klein. 1998. Computing the Edit-Distance between Unrooted Ordered Trees. In *Algorithms - ESA '98, 6th Annual European Symposium, Venice, Italy, August 24-26, 1998, Proceedings (Lecture Notes in Computer Science,*

Vol. 1461), Gianfranco Bilardi, Giuseppe F. Italiano, Andrea Pietracaprina, and Geppino Pucci (Eds.). Springer, 91–102. https://doi.org/10.1007/3-540-68530-8_8

[17] Martin Kleppmann. 2020. Moving elements in list CRDTs. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. 1–6.

[18] Martin Kleppmann, Dominic P. Mulligan, Victor B. F. Gomes, and Alastair R. Beresford. 2022. A Highly-Available Move Operation for Replicated Trees. *IEEE Transactions on Parallel and Distributed Systems* 33, 7 (2022), 1711–1724. https://doi.org/10.1109/TPDS.2021.3118603

[19] Tancred Lindholm. 2004. A three-way merge for XML documents. In *Proceedings of the 2004 ACM Symposium on Document Engineering, Milwaukee, Wisconsin, USA, October 28-30, 2004*, Ethan V. Munson and Jean-Yves Vion-Dury (Eds.). ACM, 1–10. https://doi.org/10.1145/1030397.1030399

[20] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. 2022. Peritext: A CRDT for Collaborative Rich Text Editing. *Proc. ACM Hum. Comput. Interact.* 6, CSCW2 (2022), 1–36. https://doi.org/10.1145/3555644

[21] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15.

[22] Brice Nédelec, Pascal Molli, Achour Mostéfaoui, and Emmanuel Desmontils. 2013. LSEQ: an adaptive structure for sequences in distributed collaborative editing. In *ACM Symposium on Document Engineering 2013, DocEng '13, Florence, Italy, September 10-13, 2013*, Simone Marinai and Kim Marriott (Eds.). ACM, 37–46. https://doi.org/10.1145/2494266.2494278

[23] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, and Tien N. Nguyen. 2010. Operation-Based, Fine-Grained Version Control Model for Tree-Based Representation. In *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6013)*, David S. Rosenblum and Gabriele Taentzer (Eds.). Springer, 74–90. https://doi.org/10.1007/978-3-642-12029-9_6

[24] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2016. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. In *Proceedings of the 19th International Conference on Supporting Group Work, Sanibel Island, FL, USA, November 13 - 16, 2016*, Stephan G. Lukosch, Aleksandra Sarcevic, Myriam Lewkowicz, and Michael J. Muller (Eds.). ACM, 39–49. https://doi.org/10.1145/2957276.2957310

[25] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 86–99. https://doi.org/10.1145/3009837.3009900

[26] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs, Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:12. https://doi.org/10.4230/LIPICS.SNAPL.2017.11

[27] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2006. Data consistency for P2P collaborative editing. In *Proceedings of the 2006 ACM Conference on Computer Supported Cooperative Work, CSCW 2006, Banff, Alberta, Canada, November 4-8, 2006*, Pamela J. Hinds and David Martin (Eds.). ACM, 259–268. https://doi.org/10.1145/1180875.1180916

[28] Norman Paskin. 1999. Toward unique identifiers. *Proc. IEEE* 87, 7 (1999), 1208–1227.

[29] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. 2018. Conflict-free replicated data types (CRDTs). *arXiv preprint arXiv:1805.06358* (2018).

[30] Nuno M. Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. 2009. A Commutative Replicated Data Type for Cooperative Editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*. IEEE Computer Society, 395–403. https://doi.org/10.1109/ICDCS.2009.20

[31] Hyun-Gul Roh, Myeongjae Jeon, Jinsoo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distributed Comput.* 71, 3 (2011), 354–368. https://doi.org/10.1016/J.JPDC.2010.12.006

[32] David Roundy. 2005. Darcs: distributed version management in haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2005, Tallinn, Estonia, September 30, 2005*, Daan Leijen (Ed.). ACM, 1–4. https://doi.org/10.1145/1088348.1088349

[33] David Roundy. 2009. *Darcs 2.1.0.1, Appendix A: Theory of patches.* https://www.cs.tufts.edu/comp/150GIT/archive/david-roundy/theory-patches-2009.pdf

[34] Felix Schwägerl, Sabrina Uhrig, and Bernhard Westfechtel. 2015. A graph-based algorithm for three-way merging of ordered collections in EMF models. *Sci. Comput. Program.* 113 (2015), 51–81. https://doi.org/10.1016/J.SCICO.2015.02.008

[35] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, 386–400.

[36] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs, Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 274–293. https://doi.org/10.4230/LIPICS.SNAPL.2015.274

[37] Nathan Sobo. 2022. *How CRDTs make multiplayer text editing part of Zed's DNA*. https://zed.dev/blog/crdts

[38] Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5904)*, Zhenjiang Hu (Ed.). Springer, 4–13. https://doi.org/10.1007/978-3-642-10672-9_3

[39] Tim Teitelbaum and Thomas W. Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM* 24, 9 (1981), 563–573. https://doi.org/10.1145/358746.358755

[40] Markus Voelter. 2011. Language and IDE Modularization and Composition with MPS. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 383–430.

[41] Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. *Softw. Pract. Exp.* 44, 8 (2014), 951–972. https://doi.org/10.1002/SPE.2187

[42] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2009. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*. IEEE Computer Society, 404–412. https://doi.org/10.1109/ICDCS.2009.75

[43] Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. 2024. Total Type Error Localization and Recovery with Holes. *Proc. ACM Program. Lang.* 8, POPL (2024), 2041–2068. https://doi.org/10.1145/3632910

[44] Pierre Étienne Meunier. 2024. *Version control post-Git*. FOSDEM 2024. https://archive.fosdem.org/2024/schedule/event/fosdem-2024-3423-version-control-post-git/

Received 2024-07-11; accepted 2024-11-07