Typed Image-based Programming with Structure Editing

JONATHAN EDWARDS, independent, US

TOMAS PETRICEK, University of Kent, UK

Many beloved programming systems are *image-based*: self-contained worlds that persist both code and data in a single file. Examples include Smalltalk, LISP, HyperCard, Flash, and spreadsheets. Image-based programming avoids much of the complexity of modern programming technology stacks and encourages more casual and exploratory programming. However conventional file-based programming has better support for collaboration and deployment. These problems have been blamed for the limited commercial success of Smalltalk. We propose to enable collaboration in image-based programming via types and structure editing.

We focus on the problem of schema change on persistent data. We turn to static types, which paradoxically require more schema change but also provide a mechanism to express and execute those changes. To determine those changes we turn to structure editing, so that we can capture changes in type definitions with sufficient fidelity to automatically adapt the data to suit. We conjecture that typical schema changes can be handled through structure editing of static types.

That positions us to tackle collaboration with what could be called version control for structure editing. We present a theory realizing this idea, which is our main technical contribution. While we focus here on editing types, if we can extend the approach to cover the entire programming experience then it would offer a new way to collaborate in image-based programming.

Additional Key Words and Phrases: structure editing, operational transformation, version control, schema change, image based programming

1 INTRODUCTION

Our elders tell stories of the golden age of programming in Smalltalk and LISP machines. These self-contained programming worlds use one language throughout, with all code and data persisted in an *image*[20, 36]. Image-based programming can be much simpler than our modern stack of technologies, each with their own specialized languages and data models. In an image there is no need for a distinct operating system, a file system, a shell language, a build language, a database, and none of the adapters between them. One language within one development environment is used for everything. Image-based programming is thought to encourage *exploratory programming*[24, 40]. Other beloved programming systems are also forms of image-based programming: HyperCard[21], Flash, and spreadsheets are all self-contained programming worlds persisting code and data together into a programmable document – an image by another name.

Unfortunately collaboration and deployment is harder with image-based programming than with conventional file-based programming. These problems have been blamed for the limited commercial success of Smalltalk[11]. Collaboration in file-based programming is done with *version control systems* like git[12]. But adapting that approach to images, like Smalltalk Envy[22, 29], is complicated and clashes with the native style of development. After all, it would be unsatisfying to solve the collaboration problem by importing the complexity of file-based programming.

To develop a new approach to collaborating with images we focus on a propitious special case: *schema change*, which is the problem of adapting persistent data to match code changes. Version

control systems cannot even comprehend schema change, for data and semantics is outside their purview. It is left as a problem for other tools, like Rails Migrations[4], that must be separately coded and then manually coordinated with code changes. From the perspective of image-based programming, where code and data live together harmoniously, it is only natural to expect that collaboration tools should coordinate changes to both code and data. Schema change also serves as a useful entry point because it arises even in solo image programming, allowing us to start in that simpler context.

We use static types. Dynamically typed systems like Smalltalk can handle simple schema changes, such as adding a field to a class, but not much more. General purpose solutions as in Gemstone[38] can be complex and code-intensive.

We claim that, ironically, static typing simplifies schema change. In a dynamically typed language any field can hold any value, so who is to say that value must change? It's left up to hand-written code. Putting a static type on a field is declarative: it forces a change to the value when the type changes, and tell us the acceptable range of results. The problem is reduced to inferring the appropriate data changes from a change to the type definition. That is a problem more amenable to tooling. To simplify the problem we will start with a clean slate and defer the complexities of retrofitting our approach into existing systems.

1.1 Schema Change with Abstract Structure Editing

Consider editing a record type to rename and reorder a field. If, like version control systems, we look only at the textual changes to the definition then we can't distinguish that case from deleting the field and inserting a new one with a new name. But that distinction matters for whether the field's data should be moved or deleted. Text differencing loses too much information.

To determine schema changes we use *structure editing*[9, 36, 39]. Instead of defining types with textual syntax we store them as structured values living in the image, and directly manipulate them with a GUI that performs high-level edit operations. For the example of a record type there would be distinct edit operations to insert, delete, move, and rename fields. By capturing type edits at this fine level of granularity we can perform the correct schema changes to the data, knowing to move and rename it rather than delete it.

Structure editing has been justly criticized for lacking the fluidity of text editing. But note that many image-based programming systems already use structure editing to some extent, editing text only at the finer levels of structure like Smalltalk methods and spreadsheet cells. In this paper we abstract away from the user interface issues of structure editing and focus on the semantics: what are the edits, how do they work, and how are they managed? A GUI structure editor provides affordances for these edits, but they might also be surfaced in a vi-style editor or a REPL or an API. We defer such interfaces to future work.

We conjecture that typical schema changes can be handled through structure editing of static types. Given that, the problem of collaborative schema change is reduced to the problem of collaborative structure editing. That immediately suggests collaborative editing systems like Google Docs, which uses the theory of Operational Transformation (OT)[17, 35]. However that isn't the kind of collaboration we need: rather than synchronizing distributed replicas of the same image, we need to support images that are long-lived variants of each other, display their differences, and then allow selected differences to be migrated between them. What we want is a workflow like textual version control, but for structure editing. We call this **version control for structure editing**. To provide this we adapt the theory of OT to the requirements of version control. That is the main technical contribution of this paper, which we now present.
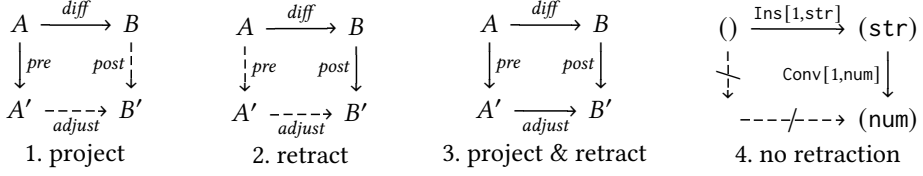
Fig. 1. Commutative diagrams for project and retract operations.

2 VERSION CONTROL FOR STRUCTURE EDITING

In this section we present the theory of our approach on the simplest possible domain: a single tuple of atomic types. The type of a *document* (our preferred term for an image) is a tuple of atomic types $(T_1, \ldots T_n)$ where $T_i \in \{\text{num}, \text{str}, \text{bool}, \text{del}\}$. The type del marks a term as deleted but retains it as a *tombstone*[33]. We define a minimal set of edit operations:

$\text{Ins}[i, t]$	Insert type t at index i, shifting indexes $\geq i$ right.
$\text{Conv}[i, t]$	Convert index i to have type t.
$\text{Move}[i, j]$	$i \neq j$. Set index i to have the type and value of index j and delete j.
Id	Does nothing.

In section 2.3 we present a precise semantics of edits including values. Until then we simplify the discussion by focusing only on changes to types.

Version control is about tracking the differences between variant documents, and migrating changes between them. The simplest situation is where document B is produced from document A with a single edit *diff*. A change is then made in the original document A and we want to be able to migrate this change into the derived document B. More formally, say that we have an edit *pre* that modifies A into A'. We want to determine the edit *post* modifying B (into B') that "does the same thing" [1]. Since we are tracking the differences between documents we also need to determine the edit *adjust* that is the new difference between the modified variants A' and B'. The function project produces *post* and *adjust* subject to this constraint:

PROPOSITION 1. $\text{project}(pre, diff) = (post, adjust) \implies post \circ diff = adjust \circ pre$

which is expressed by the commutative diagram 1 in Figure 1. The solid arrows indicate the arguments to project and the dashed arrows are the results. Note that unlike in Category Theory, our commutative diagrams are oriented, so rotation or reflection changes their meaning.

Inversely we can start with *post* and determine *pre* and *adjust* with the function retract in diagram 2 in Figure 1 above that expresses the constraint:

PROPOSITION 2. $\text{retract}(post, diff) = (pre, adjust) \implies post \circ diff = adjust \circ pre$

Projection and retraction are often (but not always) inverses. When they are we state that with Diagram 3 above using all solid arrows.

Retraction is not always possible. Diagram 4 above shows that $\text{Conv}[1, \text{num}]$ cannot be retracted through $\text{Ins}[1, \text{str}]$, which created the term being converted. There is no edit that would do the same thing as the Conv. We say that the Conv edit *depends upon* the Ins edit. Diagram 4 uses crossed-out arrows to indicate an impossible retraction.

We will walk through examples to explain the essential properties of projection and retraction. Their full definition is in Appendix A. To start with, the Ins and Move operations transform the location of other edits as would be expected:

---

[1]The rules in Appendix A define what we mean by "doing the same thing", but we don't know how to define this requirement outside the theory.

$$(\texttt{num}) \xrightarrow{\texttt{Ins[1,bool]}} (\texttt{bool}, \texttt{num}) \qquad\qquad (\texttt{str}, \texttt{num}) \xrightarrow{\texttt{Move[1,2]}} (\texttt{num}, \texttt{del})$$

$$\downarrow \texttt{Conv[1,str]} \qquad\quad \downarrow \texttt{Conv[2,str]} \qquad\qquad \downarrow \texttt{Conv[2,bool]} \qquad\quad \downarrow \texttt{Conv[1,bool]}$$

$$(\texttt{str}) \xrightarrow[\texttt{Ins[1,bool]}]{} (\texttt{bool}, \texttt{str}) \qquad\qquad (\texttt{str}, \texttt{bool}) \xrightarrow[\texttt{Move[1,2]}]{} (\texttt{bool}, \texttt{del})$$
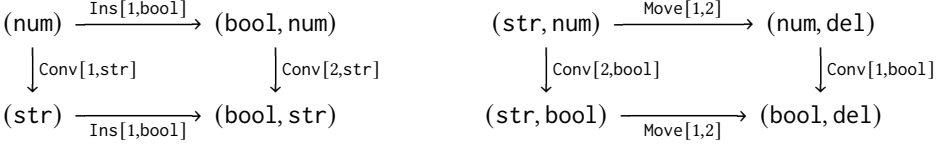
An important case is overriding of conflicting edits:

$$(\texttt{bool}) \xrightarrow{\texttt{Conv[1,str]}} (\texttt{str})$$

$$\downarrow \texttt{Conv[1,num]} \qquad\quad \downarrow \texttt{Conv[1,num]}$$

$$(\texttt{num}) \xrightarrow[\texttt{Id}]{} (\texttt{num})$$
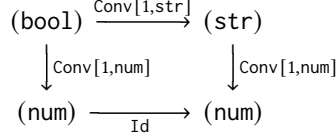
As in the above example when `Conv` operations at the same location conflict then the top edit is overridden so that the left and right edits "do the same thing". The top edit gets "grounded out" by setting the bottom edit to `Id`.[2]

This example shows that `project` and `retract` are asymmetric, meaning we sometimes cannot flip their diagrams along the diagonal (as in a matrix transpose). This asymmetry expresses that our system does not automatically resolve conflicts, but lets the user chose among possible resolutions (see 2.2). OT obtains symmetry by having timestamps preordain how to resolve conflicts.

Another important case is when duplicate operations "cancel out": if the left or right edits duplicate the top one then they make no difference when migrated to the other side, so "doing the same thing" demands they ground out to `Id`. Note how in this case `project` and `retract` are not inverses:

$$\texttt{project:} \quad \begin{array}{ccc} (\texttt{str}) & \xrightarrow{\texttt{Conv[1,num]}} & (\texttt{num}) \\ \downarrow \texttt{Conv[1,num]} & & \downarrow \texttt{Id} \\ (\texttt{num}) & \dashrightarrow[\texttt{Id}] & (\texttt{num}) \end{array} \qquad \texttt{retract:} \quad \begin{array}{ccc} (\texttt{str}) & \xrightarrow{\texttt{Conv[1,num]}} & (\texttt{num}) \\ \downarrow \texttt{Id} & & \downarrow \texttt{Conv[1,num]} \\ (\texttt{str}) & \dashrightarrow[\texttt{Conv[1,num]}] & (\texttt{num}) \end{array}$$

However except for such cases where `project` and `retract` ground out, they are inverses. In other words, if an edit "makes it through" in one direction, it will return back again restored.

PROPOSITION 3.

$$\texttt{project}(pre, \textit{diff}) = (post, adjust) \wedge post \neq \texttt{Id} \implies \texttt{retract}(post, \textit{diff}) = (pre, adjust)$$
$$\texttt{retract}(post, \textit{diff}) = (pre, adjust) \wedge pre \neq \texttt{Id} \implies \texttt{project}(pre, \textit{diff}) = (post, adjust)$$
$$\texttt{project}(\texttt{Id}, \textit{diff}) = \texttt{retract}(\texttt{Id}, \textit{diff}) = (\texttt{Id}, \textit{diff})$$

2.1 Differencing

The point of version control is to determine the differences between two variants, and migrate selected differences between them. Standard version control is based on heuristic comparison of the variants in reference to an ancestral common state[25]. We extend the standard approach in two ways. Firstly, we base our comparisons on the actual history of edits in order to be precise enough to perform schema changes. Secondly, we do not depend on finding a common ancestor. We don't want to require variants to all live in a single shared repository in which to search for an ancestor. Ancestral comparison also inaccurately identifies as differences those changes that have been duplicated in both descendants (perhaps by *cherry picking*[12]).

The point of the ancestor is to localize changes to one document or the other, for example did one document insert something or did the other one delete it? But since we are recording the edit history of each document we already have all the information we need. We can derive the "optimal

---

[2]This remains valid when we also consider values because, as discussed in section 2.3, the `Conv` operation is lossless.

ancestor" which we call the *agreement A&B* of documents A and B. This derivation also produces two sequences of edits called *differences* that reconstruct A and B from $A\&B$. The agreement is optimal in the sense that the differences are minimized, eliminating duplication.

As a starting example, consider the situation where A and B started as (num) but A inserted a bool term and B changed the num term into a str. We diagram the agreement and differences like this:

$$A = (\text{bool}, \text{num}) \xleftarrow{\text{Ins[1,bool]}} A\&B = (\text{num}) \xrightarrow{\text{Conv[1,str]}} B = (\text{str})$$

Now say that the user edits A to change the second term num to be str, as in B. In order to adjust the differences we translate the edit to A by retracting it backwards to $A\&B$ and then projecting it forwards to B, resulting in the diagram:

$$
\begin{array}{ccccc}
A = (\text{bool}, \text{num}) & \xleftarrow{\text{Ins[1,bool]}} & A\&B = (\text{num}) & \xrightarrow{\text{Conv[1,str]}} & B = (\text{str}) \\
\downarrow{\scriptstyle\text{Conv(2,str)}} & & \downarrow{\scriptstyle\text{Conv(1,str)}} & & \downarrow{\scriptstyle\text{Id}} \\
A' = (\text{bool}, \text{str}) & \xleftarrow{\text{Ins[1,bool]}} & A'\&B' = (\text{str}) & \xrightarrow{\text{Id}} & B' = (\text{str})
\end{array}
$$

Note that in the above diagram the left square is a retraction which has been mirrored to go from left to right. The right-most downward arrow, from B to B', is the translation that "does the same thing" as the edit to A. It is Id, which means that the original edit doesn't make any difference to B (because B had already made the same change). Therefore the edit is "absorbed" into the agreement, as shown in the bottom row of the diagram. Note that B has not changed but the agreement has, and there is no longer any difference between them.

But often edits do make a difference when translated, for example if A had changed the num to bool, which is different from the change B made:

$$
\begin{array}{ccccc}
A = (\text{bool}, \text{num}) & \xleftarrow{\text{Ins[1,bool]}} & A\&B = (\text{num}) & \xrightarrow{\text{Conv[1,str]}} & B = (\text{str}) \\
\downarrow{\scriptstyle\text{Conv[2,bool]}} & & \downarrow{\scriptstyle\text{Conv[1,bool]}} & & \downarrow{\scriptstyle\text{Conv[1,bool]}} \\
(\text{bool}, \text{bool}) & \xleftarrow{\text{Ins[1,bool]}} & (\text{bool}) & \xrightarrow{\text{Id}} & (\text{bool})
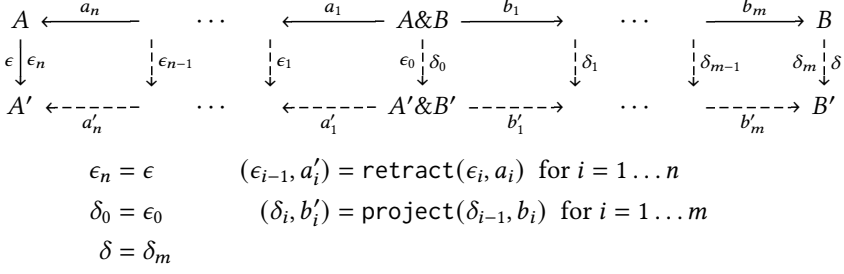\end{array}
$$

Here the right-most downwards arrow is not Id, and therefore the original edit is in fact increasing the differences between A and B, unlike the previous example. Therefore we do not adjust the agreement – instead we just append the edit to the differences of A to create A', extending the top row of the previous diagram leftward, and leaving B and the agreement unchanged, diagrammed thusly:

$$A' = (\text{bool}, \text{bool}) \xleftarrow{\text{Conv[2,bool]}} (\text{bool}, \text{num}) \xleftarrow{\text{Ins[1,bool]}} A'\&B = (\text{num}) \xrightarrow{\text{Conv[1,str]}} B = (\text{str})$$

Another case to consider is if A was to change the first term of the tuple, which does not exist in $A\&B$ or B. In that case, the edit would not retract back to the agreement, and as in the previous example would be appended to the differences of A.

To handle the general case of multiple differences we define the function translate that chains together retracts and projects. If A differs from $A\&B$ with the edit sequence $a_{1...n}$, and B likewise with $b_{1...m}$, and given an edit ϵ to A we retract it through the a edits in reverse order and then project through the b edits in forwards order. This produces an edit δ to B and adjusted difference sequences a' and b'. The whole translate is undefined if one of the retractions is undefined, meaning that ϵ depends on one of the a edits. Formally:

$$\text{translate}(\epsilon, a_{1...n}, b_{1...m}) = (\delta, a'_{1...n}, b'_{1...m}) \quad \text{where:}$$

$$A \xleftarrow{\quad a_n \quad} \quad \cdots \quad \xleftarrow{\quad a_1 \quad} A\&B \xrightarrow{\quad b_1 \quad} \quad \cdots \quad \xrightarrow{\quad b_m \quad} B$$

$$\epsilon \downarrow \epsilon_n \qquad \epsilon_{n-1} \downarrow \qquad \epsilon_1 \downarrow \qquad \epsilon_0 \downarrow \delta_0 \qquad \delta_1 \downarrow \qquad \delta_{m-1} \downarrow \qquad \delta_m \downarrow \delta$$

$$A' \xleftarrow{\quad a'_n \quad} \quad \cdots \quad \xleftarrow{\quad a'_1 \quad} A'\&B' \xdashrightarrow{\quad b'_1 \quad} \quad \cdots \quad \xdashrightarrow{\quad b'_m \quad} B'$$

$$\epsilon_n = \epsilon \qquad (\epsilon_{i-1}, a'_i) = \mathtt{retract}(\epsilon_i, a_i) \text{ for } i = 1 \ldots n$$

$$\delta_0 = \epsilon_0 \qquad (\delta_i, b'_i) = \mathtt{project}(\delta_{i-1}, b_i) \text{ for } i = 1 \ldots m$$

$$\delta = \delta_m$$

Now we can state the general algorithm for difference maintenance. If an edit ϵ to A translates into $\delta = \mathtt{Id}$ on B then the edit is *absorbed* into the agreement and the differences are adjusted into the bottom line of the diagram (leaving B unchanged). If $\delta \neq \mathtt{Id}$ or is undefined then the edit is appended to the differences of A as shown in this diagram:

$$A' \xleftarrow{\quad \epsilon \quad} A \xleftarrow{\quad a_n \quad} \quad \cdots \quad \xleftarrow{\quad a_1 \quad} A'\&B \xrightarrow{\quad b_1 \quad} \quad \cdots \quad \xrightarrow{\quad b_m \quad} B$$

The above algorithm incrementally maintains the differences and agreement between two documents as they are edited. Note that it operates on edits and never needs to inspect document states. It is straightforward to rederive the differences from scratch given the edit history of any two documents from a common ancestral state (or their total histories starting from an empty document). This is done by setting the agreement to the common ancestral state, the differences of B to its history from the ancestor, and then applying each of the edits from the history of A. It is correct because:

PROPOSITION 4. *Given two edit histories from the same starting state, the differences calculated will be the same regardless of how the two histories are interleaved.*
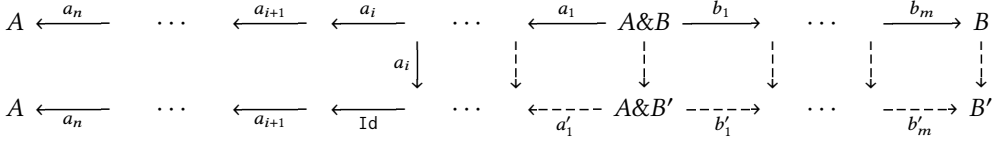
2.2 Migration

In the previous section we defined the function `translate` to derive the differences between two documents. The `translate` function is also used to migrate individual differences from one document to the other. Returning to where we left off with the last example:

$$A = (\mathtt{bool}, \mathtt{bool}) \xleftarrow{\mathtt{Conv(2,bool)}} (\mathtt{bool}, \mathtt{num}) \xleftarrow{\mathtt{Ins[1,bool]}} A\&B = (\mathtt{num}) \xrightarrow{\mathtt{Conv[1,str]}} B = (\mathtt{str})$$

Say the user wants to migrate the last difference of A, which set the second term to `bool`. We remove that edit from A's differences and then translate it through to B, producing this diagram:

$$A = (\mathtt{bool}, \mathtt{bool}) \xleftarrow{\mathtt{Conv(2,bool)}} (\mathtt{bool}, \mathtt{num}) \xleftarrow{\mathtt{Ins[1,bool]}} A\&B = (\mathtt{num}) \xrightarrow{\mathtt{Conv[1,str]}} B = (\mathtt{str})$$

$$\downarrow{\mathtt{Conv[2,bool]}} \qquad \qquad \downarrow{\mathtt{Conv[1,bool]}} \qquad \mathtt{Conv[1,bool]}\downarrow$$

$$A = \qquad\qquad (\mathtt{bool}, \mathtt{bool}) \xleftarrow[\mathtt{Ins[1,bool]}]{} A\&B' = (\mathtt{bool}) \xdashrightarrow[\mathtt{Id}]{} B' = (\mathtt{bool})$$

B has now been changed equivalently to the edit we picked to migrate, and the differences have been adjusted to remove the now agreed-upon edit. Now in this example we conveniently chose to migrate the last difference to A. In the general case we extract the subsequent differences and then append them back to the results of the translation, as in this diagram showing how to migrate the difference a_i:

$$A \xleftarrow{\ a_n\ } \quad \cdots \quad \xleftarrow{\ a_{i+1}\ } \xleftarrow{\ a_i\ } \quad \cdots \quad \xleftarrow{\ a_1\ } A\&B \xrightarrow{\ b_1\ } \quad \cdots \quad \xrightarrow{\ b_m\ } B$$

$$A \xleftarrow{\ a_n\ } \quad \cdots \quad \xleftarrow{\ a_{i+1}\ } \xleftarrow{\ \mathtt{Id}\ } \quad \cdots \quad \xleftarrow{\ a_1'\ } A\&B' \xrightarrow{\ b_1'\ } \quad \cdots \quad \xrightarrow{\ b_m'\ } B'$$

Consider what happens if $a_1 = \mathtt{Conv}(1, num)$ and $b_1 = \mathtt{Conv}(1, str)$. Migrating a_1 will override b_1, setting $b_1' = \mathtt{Id}$. We call this a *conflict*: a merge where one of the b edits is grounded out to $\mathtt{Id}$. A conflicting migration overrides differences in the opposite doc. Conflict is symmetric: if a conflicts with b then b also conflicts with a, so the user has the choice of migrating either one and overriding the other. Note that our concept of a conflict is more precise than the *merge conflict* of textual version control, which means only that changes appear to be occurring on the same text line, and often must be resolved with manual text editing. Many of our edits never conflict (like $\mathtt{Ins}$ and $\mathtt{Move}$), and conflicts that do occur are definite, and are resolved just by choosing which one wins.

PROPOSITION 5 (CONFLICT SYMMETRY). *If migrating a_i conflicts with b_j then migrating b_j will conflict with a_i.*

The key property of migration is that it reduces differences:

PROPOSITION 6 (CONVERGENCE). *Repeatedly migrating differences between two documents will terminate with them equal.*

Note however that the resultant document value can vary depending on the chosen order of migration. That order determines which of a pair of conflicting edits will dominate.

It is not possible to migrate a difference when it depends on an earlier difference – the dependency must be migrated first. In our implementation we provide the option to automatically migrate all transitive dependencies. We also support migrating all differences (equivalent to merging a branch in traditional version control) by migrating each in turn. Because we "cherry pick" at the granularity of individual edits the user interface can flexibly cluster migrations, say by time or location.

There is a technical problem with migrating $\mathtt{Ins}$ edits. Proposition 4 says we can recompute the differences between two documents from their histories, so the adjusted differences produced by the migrate function must be the same as immediately recalculating them afterwards. This fails when we migrate an $\mathtt{Ins}$ edit, because we consider all inserts to be distinct. To fix this problem we add a unique identifier to the $\mathtt{Ins}$ operation and we modify the project function to cancel out inserts with the same identifier. We left this detail out of the prior discussion for the sake of simplicity.

2.3 Value Semantics

The preceding discussion was simplified by considering only the type definitions of tuples. We will now incorporate values, and provide a precise semantics of the edit operations.

A document is a typed tuple $(v_1 : T_1 \ldots v_n : T_n)$ where $T_i \in \{\mathtt{num}, \mathtt{str}, \mathtt{bool}, \mathtt{del}\}$. The type $\mathtt{del}$ has a single value $\mathtt{null}$.

The edit operations are:

$\mathtt{Id}$	Does nothing.
$\mathtt{Ins}[i, t, p]$	Insert type t at index i, shifting indexes $\geq i$ right. p is a unique identifier.
$\mathtt{Conv}[i, t]$	Convert index i to have type t.
$\mathtt{Move}[i, j]$	$i \neq j$. Set index i to have the type and value of index j and delete j.

defined as:

$$\mathtt{Id}\,(v_1:T_1\ldots v_n:T_n) = (v_1:T_1\ldots v_n:T_n)$$

$$\mathtt{Ins}[i,t,p]\,(v_1:T_1\ldots v_n:T_n) = (v_1:T_1\ldots v_{i-1}:T_{i-1},\mathtt{null}:t,v_i:T_i\ldots v_n:T_n)$$

$$\mathtt{Conv}[i,t]\,(v_1:T_1\ldots v_n:T_n) = (v_1:T_1\ldots v_{i-1}:T_{i-1},v_i:t,v_{i+1}:T_{i+1}\ldots v_n:T_n)$$

$$\mathtt{Move}[i,j]\,(v_1:T_1\ldots v_n:T_n) = (v_1':T_1'\ldots v_n':T_n') \text{ where } v_k':T_k' = \begin{cases} v_j:T_j & k=i \\ \mathtt{null}:\mathtt{del} & k=j \\ v_k:T_k & \text{otherwise} \end{cases}$$

The rules defining projection and retraction are presented in Appendix A.

We allow a document to contain type errors. In the spirit of image-based programming, we handle errors gracefully and allow them to be corrected when convenient. We define a function conform that takes a document and produces a type-conforming version of it, in which all values either conform to their type or are the special value error. Programs can only see conforming values. This technique is related to the *non-empty holes* of Omar et al.[31, 32].

The Ins edit gives a new term the value null, representing an uninitialized value. The conform function can treat an uninitialized value as an error, or as we do in our implementation, assign a default value based on the type.

The conform function also handles type conversion. The Conv edit just sets the desired type, leaving the existing value in place, which conform converts if possible or yields error. By retaining the unconverted value we also allow conversions to compose losslessly, which occurs when we migrate conflicting conversions. Lossless conversions are needed to ensure convergence (Proposition 6).

3 PROTOTYPE IMPLEMENTATION

Will this theory work in practice? We are building a prototype implementation to find out. We have extended the minimalist theory to a more practical set of datatypes and edit operations. We have also built a GUI to explore how to visually manifest structural version control. We have extended the theory to include:

(1) Nested records and homogeneous lists, forming a tree.
(2) Edit operations are located at paths within the tree, expressed as sequences of indexes.
(3) Move carries a subtree across the tree to replace another subtree. This includes the ability to reorder lists.
(4) Edits to the type of a list are iterated over its elements.
(5) A rename operation on record fields.
(6) Differences can be *normalized*, producing a sequence of edits ordered by location in the tree.
(7) (pending) Sum types, to gain parity with standard FP datatypes.
(8) (pending) A wrap operation to replace a location the tree with a record or sum or list containing it. There is an inverse unwrap operation.

Our prototype GUI focuses on viewing the differences between two documents and performing migrations. A central problem of structure editing is to make it as "fluid" as text editing. That is not the problem we are working on here, so we edit with a command line. We use infix syntax for edit operations, with the prefixed location formatted as a dotted path of field names and list indexes.

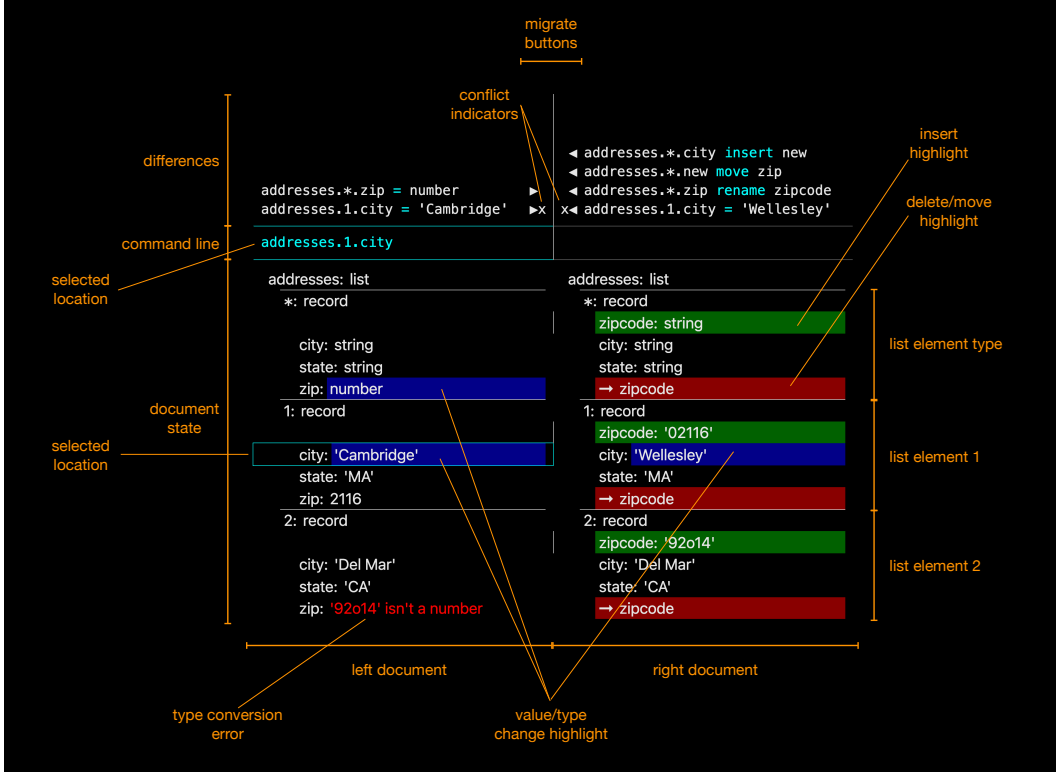Figure 2 shows an annotated screenshot. A recorded demo can be viewed at https://vimeo.com/manage/videos/631461226.

Fig. 2. Annotated screenshot of difference view

4 LIMITATIONS AND OPPORTUNITIES

This is early stage work that raises more questions than it answers:

(1) Can direct manipulation editing in a GUI compete with the fluidity of text editing?
(2) Can editing be extended to encompass code, enabling coordinated refactorings across code and data?
(3) Can complex interrelated schema changes be accommodated? Ambler&Sadalage[6] cast many such large scale schema changes as refactorings, which we hope to support as structure edits. Nevertheless in some cases it will still be necessary to write code to convert data. How does such code interact with edit migration?
(4) Can these techniques be retrofitted into existing image-based systems (Smalltalk, LISP), or even text-based ones?

Version control has become a central technology of programming, but we see the chance to enlarge its impact even further:

(1) Support a larger audience of programmers by providing a simpler and more tractable conceptual model than git [14].
(2) Images could be utilized like OS containers, so that deployment becomes version control. Deployment was a major problem in the earlier age of image-based programming.
(3) We mentioned at the outset that spreadsheets are an example of image-based programming. We would love to have our approach applied to spreadsheets. Instead of making tweaked

copies and copy-pasting between them, we could view their differences and migrate changes as needed.

Our long term vision is to use types and structure editing to bring back the benefits of image-based programming to modern programmers. We hope that we have at least incentivised future research on structure editing by identifying two heretofore overlooked benefits:

(1) Automatic schema change, and
(2) high-level version control.

5 RELATED WORK

SQL DDL[1] is the standard language for editing relational database schema. It can rename, re-order, and convert the datatype of a column. More complex database conversions can be coded in specialized languages such as Rails Migrations[4]. Schema change has been cast as a form of refactoring[6]. Versioning has been built into or added onto databases and other structured data formats[2, 3, 7, 18] and notebooks[24]. The theory of lenses has been applied to schema change on data structures[19, 28]. Closer to our focus on image-based programming is the work on schema change for Object Oriented Databases[8, 38], version control for Smalltalk[29], and Orthogonal Synchronization[10].

Research on structure editing has a long history[9, 36, 39]. One focus has been to enable non-textual *projections* of code[13, 37]. Perhaps the most successful structure editor is Scratch[34], providing an easy editing experience for beginners free of syntax errors. Scratch and successor *blocks-based* languages rely on drag-and-drop gestures, which motivated our inclusion of the Move operation. Recent research seeks to match the fluidity of text editing [23, 27] and the rigorous semantics of PL theory[31, 32].

Our work repurposes Operational Transformation (OT) [17, 35], which is the theory behind collaborative document editing as in Google Docs. OT supports collaborative editing of a single document, where the central problem is the unpredictable arrival order of edits at distributed replicas. Our need to track and selectively migrate differences between long-lived variants leads to a substantially different and more complex theory. For one thing, we give users the choice of how and when to resolve conflicts, whereas in OT timestamps preordain resolution. This introduces asymmetries into our transformations, violating the Symmetry Property of Ressel et al.[35]. The notation of commutative diagrams from Category Theory has proven essential to handle the complexity arising, ironically, from the non-commutativity of edits.

The Categorical Theory of Patches [30] implemented in Pijul[5] presents a theory of textual version control based on line-oriented text edits called patches. Intriguingly, a merge is modelled as a categorical pushout, diagrammed like our project function, and our retract function looks like a pullback. However the asymmetries introduced by overriding conflicts don't satisfy the definitions of pushout and pullback. We don't know what to make of this and welcome any insights from Category Theorists.

Our prior work on Subtext [15, 16] assigned permanent parent-unique IDs to tree nodes, forming stable paths. However differencing/migrating movement and wrapping/unwrapping operations became complex and confusing. Ultimately we felt forced to adopt an OT-like approach of transforming edit histories in order to have some confidence of correctness. Folk wisdom has it that OT is difficult to get right, but so far we have found the implementation guided by the theory to be robust. Note that unique IDs had to be incorporated into Ins operations to support Proposition 4, so the status of IDs is still unclear. It would be interesting to see if recent CRDT advances [26] could provide an alternative solution.
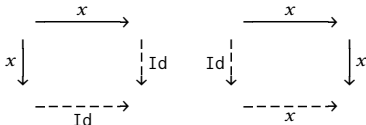
6 ACKNOWLEDGEMENTS

REFERENCES

[1] 2021. ALTER TABLE, SQL Language Reference. https://docs.oracle.com/en/database/oracle/oracle-database/21/sqlrf/ALTER-TABLE.html#GUID-552E7373-BF93-477D-9DA3-B2C9386F2877

[2] 2021. DBmaestro: DevOps for Database. https://db.dbmaestro.com/

[3] 2021. Dolt. https://www.dolthub.com/

[4] 2021. Migrations. https://guides.rubyonrails.org/v3.2/migrations.html

[5] 2021. Pijul, a sound and fast distributed version control system based on a mathematical theory of asynchronous work. https://pijul.org/

[6] S.W. Ambler and P.J. Sadalage. 2006. *Refactoring Databases: Evolutionary Database Design.* Addison Wesley, USA.

[7] Attic-Labs. 2021. Noms. https://github.com/attic-labs/noms

[8] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. 1987. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD Rec.* 16, 3 (Dec. 1987), 311–322. https://doi.org/10.1145/38714.38748

[9] D.R. Barstow, S.G. Guty, H.E. Shrobe, and E. Sandewall. 1984. *Interactive Programming Environments.* McGraw-Hill, USA.

[10] Gilad Bracha. 2005. Objects as Software Services. http://bracha.org/objectsAsSoftwareServices.pdf

[11] Gilad Bracha. 2020. Bits of History, Words of Advice. https://gbracha.blogspot.com/2020/05/bits-of-history-words-of-advice.html

[12] Scott Chacon and Ben Straub. 2014. *Pro Git* (2nd ed.). Apress, USA.

[13] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley Publishing Co., USA, Chapter 11.

[14] Santiago Perez De Rosso and Daniel Jackson. 2016. Purposes, Concepts, Misfits, and a Redesign of Git. *SIGPLAN Not.* 51, 10 (Oct. 2016), 292–310. https://doi.org/10.1145/3022671.2984018

[15] Jonathan Edwards. 2005. Subtext: Uncovering the Simplicity of Programming. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05).* Association for Computing Machinery, New York, NY, USA, 505–518. http://www.subtext-lang.org/OOPSLA05.pdf

[16] Jonathan Edwards. 2006. *First Class Copy & Paste.* Technical Report MIT-CSAIL-TR-2006-037. MIT. http://hdl.handle.net/1721.1/32980

[17] C. A. Ellis and S. J. Gibbs. 1989. Concurrency Control in Groupware Systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, USA) *(SIGMOD '89).* Association for Computing Machinery, New York, NY, USA, 399–407. https://doi.org/10.1145/67544.66963

[18] Paul Fitz. 2021. daff. https://paulfitz.github.io/daff/

[19] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems* 29, 3 (May 2007), 17. https://doi.org/citation.cfm?doid=1232420.1232424

[20] Adele Goldberg. 1984. *SMALLTALK-80: The Interactive Programming Environment.* Addison-Wesley Longman Publishing Co., Inc., USA.

[21] D. Goodman and W. Atkinson. 1987. *The Complete HyperCard Handbook.* Bantam Books, USA.

[22] Paul Hammant. 2017. Smalltalk Envy. https://paulhammant.com/2017/09/01/smalltalk-envy/

[23] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: A Lightweight User Interface for Structured Editing. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18).* Association for Computing Machinery, New York, NY, USA, 654–664. https://doi.org/10.1145/3180155.3180165

[24] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. *Variolite: Supporting Exploratory Programming by Data Scientists.* Association for Computing Machinery, New York, NY, USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[25] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. 2007. A Formal Investigation of Diff3. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, V. Arvind and Sanjiva Prasad (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 485–496.

[26] Martin Kleppmann, Dominic P. Mulligan, Victor B. F. Gomes, and Alastair Beresford. 2021. A highly-available move operation for replicated trees. *IEEE Transactions on Parallel and Distributed Systems* (2021), 1–1. https://doi.org/10.1109/TPDS.2021.3118603

[27] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. 2015. Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (London, United Kingdom) *(WiPSCE '15).* Association for Computing Machinery, New York, NY, USA, 29–38. https:
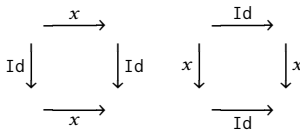
//doi.org/10.1145/2818314.2818331

[28] Geoffrey Litt, Peter van Hardenberg, and Henry Orion. 2020. Project Cambria: Translate your data with lenses. https://www.inkandswitch.com/cambria.html

[29] Vikas Malik. 1997. Smalltalk Envy. http://www.faqs.org/faqs/smalltalk/ENVY-faq/

[30] Samuel Mimram and Cinzia Di Giusto. 2013. A Categorical Theory of Patches. *Electronic Notes in Theoretical Computer Science* 298 (Nov 2013), 283–307. https://doi.org/10.1016/j.entcs.2013.09.018

[31] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *PACMPL* 3, POPL (2019). https://doi.org/10.1145/3290327

[32] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*.

[33] Gerald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. 2006. Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In *2006 International Conference on Collaborative Computing: Networking, Applications and Worksharing*. 1–10. https://doi.org/10.1109/COLCOM.2006.361867

[34] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[35] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. 1996. An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work* (Boston, Massachusetts, USA) *(CSCW '96)*. Association for Computing Machinery, New York, NY, USA, 288–297. https://doi.org/10.1145/240080.240305

[36] Erik Sandewall. 1978. Programming in an Interactive Environment: The "Lisp" Experience. *ACM Comput. Surv.* 10, 1 (March 1978), 35–71. https://doi.org/10.1145/356715.356719

[37] JetBrains s.r.o. 2021. MPS: The Domain-Specific Language Creator. https://www.jetbrains.com/mps/

[38] GemTalk Systems. 2015. Class versions and Instance Migration. https://downloads.gemtalksystems.com/docs/GemStone64/3.2.x/GS64-ProgGuide-3.2/10-ClassHistory.htm

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

[40] J. Trenouth. 1991. A Survey of Exploratory Software Development. *Comput. J.* 34, 2 (01 1991), 153–163. https://doi.org/10.1093/comjnl/34.2.153 arXiv:https://academic.oup.com/comjnl/article-pdf/34/2/153/1400604/340153.pdf
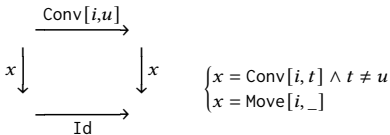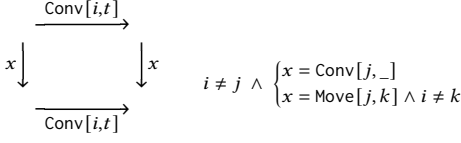
A PROJECTION AND RETRACTION RULES

Equal edits cancel out:

Id is a fixpoint:

Conv is overridden by conflicting changes:
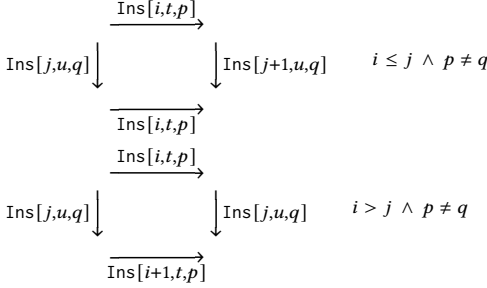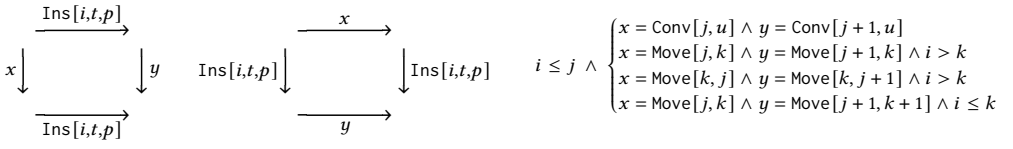

$$\begin{cases} x = \text{Conv}[i, t] \land t \neq u \\ x = \text{Move}[i, \_] \end{cases}$$

Conv passes through independent edits:

$$
\begin{array}{ccc}
x & \xrightarrow{\text{Conv}[i,t]} & \\
\downarrow x & & \downarrow x \\
& \xrightarrow{\text{Conv}[i,t]} &
\end{array}
\qquad i \neq j \;\wedge\; \begin{cases} x = \text{Conv}[j,\_] \\ x = \text{Move}[j,k] \wedge i \neq k \end{cases}
$$

Ins doesn't affect leftward edits:

$$
\begin{array}{ccc}
x & \xrightarrow{\text{Ins}[i,t,p]} & \\
\downarrow x & & \downarrow x \\
& \xrightarrow{\text{Ins}[i,t,p]} &
\end{array}
\qquad
\begin{array}{ccc}
& \xrightarrow{x} & \\
\text{Ins}[i,t,p]\downarrow & & \downarrow\text{Ins}[i,t,p] \\
& \xrightarrow{x} &
\end{array}
\qquad i > j \;\wedge\; \begin{cases} x = \text{Conv}[j,\_] \\ x = \text{Move}[j,k] \wedge i > k \end{cases}
$$
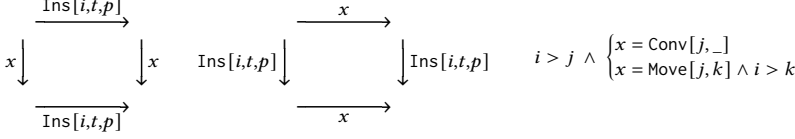
Ins shifts non-leftward edits rightward:

$$
\begin{array}{ccc}
x & \xrightarrow{\text{Ins}[i,t,p]} & \\
\downarrow y & & \downarrow x \\
& \xrightarrow{\text{Ins}[i,t,p]} &
\end{array}
\qquad
\begin{array}{ccc}
& \xrightarrow{x} & \\
\text{Ins}[i,t,p]\downarrow & & \downarrow\text{Ins}[i,t,p] \\
& \xrightarrow{y} &
\end{array}
\qquad i \leq j \;\wedge\;
\begin{cases}
x = \text{Conv}[j,u] \wedge y = \text{Conv}[j+1,u] \\
x = \text{Move}[j,k] \wedge y = \text{Move}[j+1,k] \wedge i > k \\
x = \text{Move}[k,j] \wedge y = \text{Move}[k,j+1] \wedge i > k \\
x = \text{Move}[j,k] \wedge y = \text{Move}[j+1,k+1] \wedge i \leq k
\end{cases}
$$

$$
\begin{array}{ccc}
& \xrightarrow{\text{Ins}[i,t,p]} & \\
\text{Ins}[j,u,q]\downarrow & & \downarrow\text{Ins}[j+1,u,q] \\
& \xrightarrow{\text{Ins}[i,t,p]} &
\end{array}
\qquad i \leq j \;\wedge\; p \neq q
$$

$$
\begin{array}{ccc}
& \xrightarrow{\text{Ins}[i,t,p]} & \\
\text{Ins}[j,u,q]\downarrow & & \downarrow\text{Ins}[j,u,q] \\
& \xrightarrow{\text{Ins}[i+1,t,p]} &
\end{array}
\qquad i > j \;\wedge\; p \neq q
$$

Cannot retract an edit at a location through the insert of that location. Not even an insert at that location, because that would not be reversible by a projection.

$$
\begin{array}{ccc}
& \xrightarrow{\text{Ins}[i,\_,p]} & \\
\not\downarrow & & \downarrow x \\
& \dashrightarrow\!\!\!\not\!\!\!\dashrightarrow &
\end{array}
\qquad
\begin{cases}
x = \text{Ins}[i,\_,q] \wedge p \neq q \\
x = \text{Conv}[i,\_] \\
x = \text{Move}[i,\_] \\
x = \text{Move}[\_,i]
\end{cases}
$$
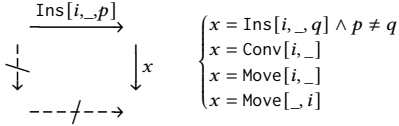
Move moves source to target:

$$
\begin{array}{ccc}
x & \xrightarrow{\text{Move}[i,j]} & \\
\downarrow y & & \downarrow x \\
& \xrightarrow{\text{Move}[i,j]} &
\end{array}
\qquad
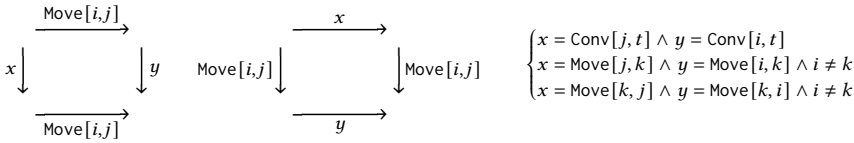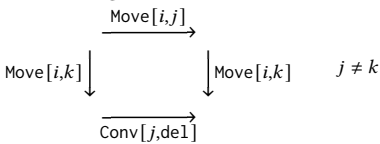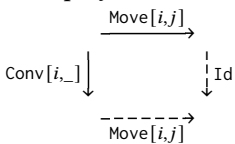\begin{array}{ccc}
& \xrightarrow{x} & \\
\text{Move}[i,j]\downarrow & & \downarrow\text{Move}[i,j] \\
& \xrightarrow{y} &
\end{array}
\qquad
\begin{cases}
x = \text{Conv}[j,t] \wedge y = \text{Conv}[i,t] \\
x = \text{Move}[j,k] \wedge y = \text{Move}[i,k] \wedge i \neq k \\
x = \text{Move}[k,j] \wedge y = \text{Move}[k,i] \wedge i \neq k
\end{cases}
$$

Conflicting Move edits delete the overridden Move's source:

$$
\begin{array}{ccc}
& \xrightarrow{\text{Move}[i,j]} & \\
\text{Move}[i,k]\downarrow & & \downarrow\text{Move}[i,k] \\
& \xrightarrow{\text{Conv}[j,\text{del}]} &
\end{array}
\qquad j \neq k
$$

Can't project Conv through Move to its target, because it would retract to the source of the Move.

$$
\begin{array}{ccc}
& \xrightarrow{\text{Move}[i,j]} & \\
\text{Conv}[i,\_]\downarrow & & \downarrow\text{Id} \\
& \dashrightarrow[\text{Move}[i,j]] &
\end{array}
$$

Move passes through edits not to target or source:

$$
\begin{array}{ccc}
& \xrightarrow{\;\text{Move}[i,j]\;} & \\
x\Big\downarrow & & \Big\downarrow x \\
& \xrightarrow{\;\text{Move}[i,j]\;} &
\end{array}
\qquad
i \neq k \;\wedge\; j \neq k \;\wedge\;
\begin{cases}
x = \text{Conv}[k, \_] \\
x = \text{Move}[k, l] \wedge i \neq l \wedge j \neq l
\end{cases}
$$