Denicek: Computational Substrate for Document-Oriented End-User Programming

Anonymous Author(s)

Abstract

User-centric programming research gave rise to a variety of compelling programming experiences, including local-first collaborative editing, programming by demonstration, incremental recomputation, schema change control, end-user debugging and concrete programming. Those experiences advance the state of the art of end-user programming, but they are hard to implement on the basis of established programming languages and system.

We contribute Denicek, a computational substrate that simplifies the implementation of the above programming experiences. Denicek represents a program as a series of edits that construct and transform a document consisting of data and formulas. It provides three operations on edit histories: edit application, merging of histories and conflict resolution. By composing these operations, we can easily implement a range of programming experiences.

We present the architecture of Denicek, discuss key design considerations and elaborate the implementation of a variety of programming experiences. To evaluate the proposed substrate, we use Denicek to develop an innovative interactive data science notebook system. The case study shows that the Denicek computational substrate provides a suitable basis for the design of rich, interactive end-user programming systems.

Keywords

Programming Systems, Computational Substrate, End-User Programming, Programming by Demonstration, Local-First Software

ACM Reference Format:
Anonymous Author(s). 2025. Denicek: Computational Substrate for Document-Oriented End-User Programming. In *Proceedings of (Conference acronym 'XX).* ACM, New York, NY, USA, 16 pages. https://doi.org/XXXX.XXXX

1 Introduction

A computational substrate defines the structures with which programs are constructed, how the progam state is represented and how it evolves [38]. The choice of a substrate affects what programming experiences can be readily supported by a system. For example, object-oriented programming has been historically linked to graphical user interfaces [45], while representing programs as lists enabled Lisp to become a language laboratory [94].

In principle, any programming experience can be developed on top of any computational substrate. However, a suitable program representation can eliminate much of the complexity of implementing interesting programming experiences. For example, the reflective capabilities of Smalltalk make it easy to build rich debugging tools [84] that are difficult to implement for C/C++ [46, 47].

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
2025. ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXX.XXXX

Programming Experiences. In this paper, we describe a programming substrate that makes it easy to develop diverse programming experiences [68]. Those include:

- *Local-First Collaboration.* Multiple users should be able to concurrently modify a single document and automatically merge their changes, preferrably without a central server [50, 62].
- *Programming by Demonstration.* Allow users to construct simple programs by enacting the steps of the expected behavior using concrete examples [13, 58].
- *Incremental Recomputation.* When a part of a document changes, formulas whose result depends on it are invalidated and, possibly, automatically recomputed [34, 65, 96].
- *Schema Change Control.* When the user evolves the structure of the document, affected data and formulas should automatically co-evolve to match the new structure [25, 63].
- *End-User Debugging.* The user should be able to ask provenance questions [12] – why a computation resulted in a particular value and what inputs contributed to the result [54].
- *Concrete Programming*. It should be possible to reuse parts of program logic, or formulas, without introducing abstractions, that is, program against concrete values [22, 24].

Two-Stage Methodology. The technical focus of this paper fits within the interior mode of design science research [2]. To *design* Denicek, we identify six *formative examples* – simple programming tasks that manifest one or more of the desired programming experiences (§A). Using those examples, we co-design the Denicek substrate and Webnicek (§3), a simple web-based end-user programming environment built directly on top of the substrate. Although Webnicek can be used to complete end-user programming tasks, it is optimized for developing the underlying substrate rather than for usability.

To *evaluate* Denicek, we use the substrate to develop Datnicek (§7), an interactive data science notebook system inspired by existing environments [18, 43]. We use the second stage to evaluate suitability of the Denicek substrate for the development of end-user programming systems and report the results of our evaluation (§8).

Substrate. The Denicek substrate brings together two central design ideas. First, it represents programs as document trees consisting of nodes that can represent data, formulas, evaluated results, as well as static content. Second, Denicek does not store the document tree itself, but instead, maintains a sequence of edit operations through which the tree was constructed and transformed.

The substrate then provides three primitive operations for working with sequences of edits. First, it can apply a series of edits to reconstruct the document. Second, it can merge two diverging edit histories. Finally, it can detect conflicts when merging histories and, for example, remove conflicting edits from one branch.

The key insight that we present in this paper is that many compelling programming experiences can be implemented by leveraging the uniform document representation alongside with a suitable composition of the three primitive operations.

Changing data and formulas is done using the same primitive edit operations that manipulate the document structure. A user-interface may provide a specialized editor, but still trigger the primitive edits behind the scenes. However, interacting with elements in the document, such as a entering text in a textbox can also generate a document edit that can be merged or checked for conflicts (§5.3).

As we will see, past edits that demonstrate an operation done to the document can be recorded, allowing programming by demonstration (§5.2). Replaying such recorded edits is implemented using the merging operation, which means that recorded operations continue working even if the document structure later evolves. Moreover, structural changes to the document can be merged with concurrent data edits (§5.1). Evaluation of formulas also generates document edits (§5.4). If the evaluated edits conflict with manual edits done later by the user, the evaluated edits are removed, implementing an incremental recomputation (§5.5).

Contributions. The structure and contributions of this paper are:

- We present the Denicek substrate (§4) and provide a detailed description of its document representation, edit operations and the key three operations for working with edit histories.
- We illustrate a range of end-user programming experiences supported in Webnicek, a simple web-based prototype programming system (§3), and discuss how the experiences are implemented using the Denicek substrate (§5).
- We document important design decisions, alternatives and limitations (§6). The analysis shows that the desired functionality requires a careful choice among interconnected design options.
- We present Datnicek, an end-user data exploration system built on top of Denicek (§7) and evaluate the degree to which the Denicek substrate simplifies such development (§8).

To enable others build on top of Denicek, we share our compact and documented source code at: https://github.com/removed/for/review

2 Background

Denicek aims to support a class of systems associated with end-user programming [68], notational freedom and self-sustainability [37], liveness, interactivity and richness [35, 86]. We see programming more as interacting with a medium or a substrate [44, 51] than writing text [26]. We focus on systems that allow gradual progression from a user to a developer, as envisioned in Smalltalk [85] and use the term end-user programming loosely to refer to a part of this spectrum, also including spreadsheet systems and notebooks for data science. We first review related systems and their technical design (§2.1) before discussing programming experiences (§2.2).

2.1 Programming Systems and Substrates

Programming Systems and Substrates. A number of systems illustrate the qualities Denicek aims to support and have a related internal structure. Subtext, BootstrapLab and Infra [21, 31, 38] use structured document-based program representation and provide some of the desired programming experiences on top of this representation. Many of those design ideas can be traced back to Boxer [17], which introduced the *naive realism* principle (what the user sees is all there is) that we follow in Webnicek.

To indicate that Denicek is intended as an underlying infrastructure on top of which programming systems can be built, we use the term *computational substrate*, which also dates back to Boxer [16] and is closely related to the notion of dynamic media introduced by Kay and Goldberg [44]. Webstrates [51] is a substrate based on synchronization of documents (but without edit histories) that has been use as the basis for multiple programming systems [8, 82].

Edit Histories and Merging. Manipulating programs through semantically meaningful edits is a technique used by structure editors [5, 32, 96]. The langauge of edits has been captured formally as an *edit calculus* [74] and edits have also been used as the basis of live programming environments [99].

Merging of edits is most frequently done in version control systems such as git. Systems based on a more rigorous design such as Pijul [102] delay merging to a later point by using a graph or a lattice [89]. In the context of programming environments, Grove [3] uses a commutative patches, with a graph structure similar to Pijul, as the basis for a collaborative structure editor.

More generally, merging of edits can be based on the operational transform approach [15], where edit conflicts are reconciled, or conflict-free representation of edits (CRDTs) [50, 62]. In both approaches, supporting complex edits on tree structures such as JSON remains a challenge [14, 41]. Mergeable replicated data types (MRDTs) [42] can merge updates provided there is a suitable relational representation of the data. Recent work on MySubstrates and Grove [3, 52] has been based on CRDTs, whereas Edwards et al. [23] use operational transformations.

2.2 Programming Experiences

We review a range of compelling programming experiences explored in recent research, focusing on those supported by Denicek.

Local-First Collaboration. Since the early collaborative programming environments such as Collabode [28], real-time collaboration has became widely used, if not always without challenges [95]. Merging concurrent edits is one such challenge. In addition to CRDT-based approaches [52], conflicts arising during collaboration have been solved through fine-grained locking [100].

Programming by Demonstration. Earliest programming by demonstration systems used the paradigm for tasks ranging from graphics and user interfaces to general-purpose programming [13, 92]. Wrangler [43] showed the effectivity of the paradigm for data cleaning, whereas more recent uses range from augmented reality prototyping [58] and web automation [11] to end-user software customizations [60, 61]. A more general class of demonstrational interfaces [69] also includes programming by example, used for example to infer data transformations in spreadsheets [29]. Demonstrational systems can be used to directly perform actions, but also to generate code that captures the interaction, as done e.g. in WREX [18].

Incremental Recomputation. Interactive programming systems with live previews [65, 78] have been attempting to update the previews

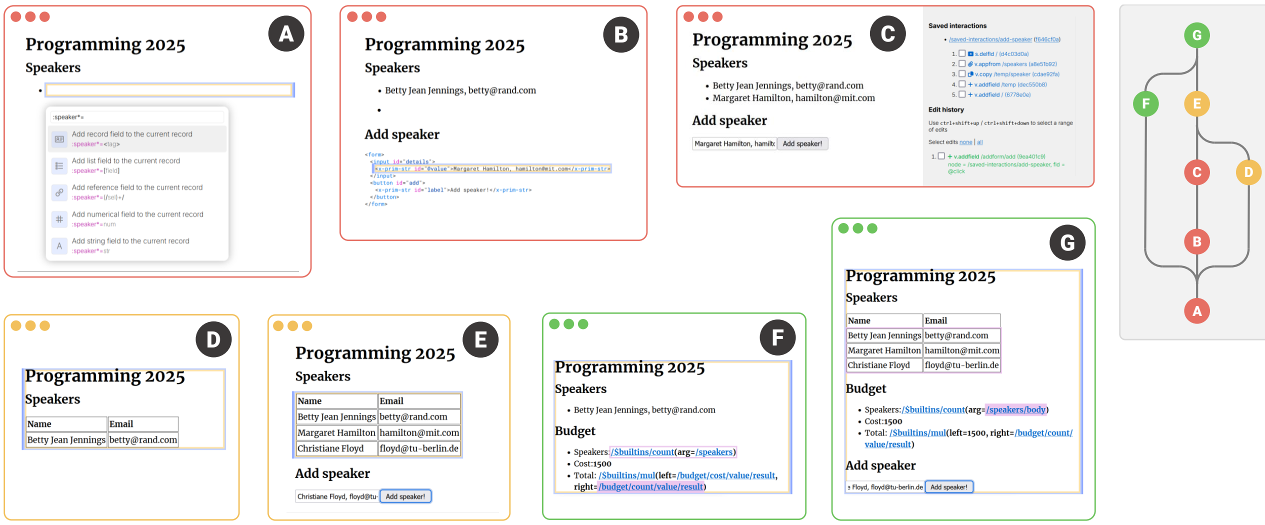

Figure 1: Organizing a conference using Denicek. The Walkthrough shows construction of a user interface for adding speakers (A, B, C); refactoring of the list and merging edits (D, E); and formulas with schema and code co-evolution (F, G).

without full recomputation at least since the pioneering work on the Cornell Program Synthesizer [96]. Outside live programming, more work has been focused on updating computation when data change, although this can also be source code of a program passed to an adaptive interpreter [1]. Incremental recomputation is also a concern in notebook systems where cells can be run out of order [91]. This can be addressed by maintaining a dependency graph [55, 81] that also allows incremental recomputation on code change.

Schema Change Control. Schema change control is concerned with adapting data and code to reflect changes in schema. The problem is well-studied in the context of databases [9], although only few systems also automatically adapt database queries [101]. However, the problem is starting to be recognized in programming systems research [25, 63] as well as live programming [4] where state needs to be preserved during program editing.

End-User Debugging. Non-programmers also need to be able understand and debug their programs [49]. Systems such as Whyline and Probe Log [53, 54, 56] record information about program execution to let users analyse why they see a particular result, whereas displaying intermediate steps can aid understanding of data science pipelines [90]. More generally, such functionality can leverage provenance tracking [12] and program slicing [76, 87]. The same infrastructure can also be used to build linked visualizations [77].

Concrete Programming. Programming can be simplified by working with concrete values instead of abstractions, an idea pioneered by the prototype-based programming language Self [98]. Self uses prototypes at the object level. At the expression level, similar functionality can be provided by managed copy & paste. Subtext [22, 24] treats this mechanism as central, whereas other systems view tracking of copy & paste as an additional editor featrue [36, 97]. Notably, copy & paste has been also tracked in spreadsheets [33] and Gridlets [40] offer an appealing alternative design.

Other Programming Experiences. Several experiences not directly addressed in this paper are worth further investigation. Projectional editors such as Lorgnette [27], live literals [73], projection boxes [59] and data detectors [72] allow visualization or editing of aspects of programs through a user interface. The problem of making live and rich editors compositional is addressed by Engraft [34]. Programming environments may also integrate live examples [84] and a variety of AI assistantce tools [7, 66, 80].

3 Walkthrough

In this section, we use the web-based prototype Webnicek to illustrate programming experiences enabled by the Denicek substrate (§5). Webnicek is based on a structure editor that supports navigation in the document and issuing of edit commands. We follow two formative examples (§A) related to conference planning.

A *Adding a Speaker.* The user starts with an empty document, which is represented as a record. They add a field for each heading and a list `<ul>`, represented by a field named `speakers`. They use the command toolbox to add the first speaker.

B *Creating a User Interface.* To simplify adding of further speakers, the user creates a textbox and a button. They enter new speaker details, add a new `<li>` element to the list and copy the speaker details from the textbox into the new element using source view.

C *Abstracting the Interaction.* After adding the speaker from the textbox, the user opens history view, selects edits that added the speaker and saves those as the `add-speaker` interaction. They attach this as a handler for the `click` event of the button.

> **Programming by Demonstration.** Denicek implements programming by demonstration (§5.2) by letting the user to save and replay past interactions, either directly or by attaching them to a UI event.

D *Refactoring Document Structure.* Another user starts with the initial version of the document and turns the list into a table. This is done by invoking a series of commands that change tags, wrap elements, copy and transform values.

E *Merging Edits.* The two document versions (C and D) can be automatically merged. The refactoring is applied to all existing speakers. New speakers added using the "Add speaker!" button are also automatically added in the new format.

> **Local-First Collaboration.** Denicek's merging reapplies edits from another branch on top of the current history (§5.1). Merging is asymmetric, but the order does not matter in the above scenario. The same merging operation is used when handling user interaction (§5.3).

F *Adding Budget Calculation.* A third user adds budget calculation to the initial document. Formulas are represented as special `x-formula` nodes whose arguments are other nodes or references to other nodes or formulas (highlighted on mouse hover).

> **Incremental Recomputation.** Evaluating a formula yields additional edits that augment the document with the result. Those edits are kept at the top of the document history and are removed in case of a conflict (§5.5), providing incremental recomputation.

G *Merging Formulas.* When the budget calculation is merged with the other edits, references in formulas are automatically updated to point to the new list. Adding new speaker via the "Add speaker!" button invalidates the evaluated result.

> **Schema Change Control.** The substrate understands references in the document and updates them when applying structural edits (§5.6). Evaluation can replace formulas with values, but also augment them to enable end-user debugging via provenance analysis (§5.7).

4 The Denicek Substrate

Denicek represents programs as sequences of edits that construct and transform a computational document. In this section, we describe the structure of documents and edits, as well as the operations that form the backbone of the system and are used to implement the end-user programming experiences discussed in §5.

4.1 Selectors, Documents and Edits

A computational document is a tree, consisting of four kinds of nodes (Fig. 2). Non-leaf nodes are records or lists. Both of these are named with a *tag* as in HTML. Record fields have unique? names. Lists contain elements of the same type and assign a unique index to each element It seems that these indexes are meant to be unique IDs, in which case it would be clearer to call them that. Leaf nodes can be primitive values (strings and numbers) or references to a location in the document tree. References can be relative or absolute. Both kinds can appear in a document (reference nodes); absolute references are used in edits (to denote a target node). Why are relative references needed? I only use absolute references.

A reference is represented as a sequence of selectors (Fig. 2). The document model assumes that lists are homogeneous and records heterogeneous, and so the Any selector makes it possible to refer to all children of a list, but there is no way to refer to all children of a record. Note that Denicek does not use implicit numerical indices for lists. The index of a new item has to be supplied explicitly. The reasoning behind this design choice is discussed in §6.

Selector	Notation	
Parent	`..`	Refers to a parent of a node
Field	`field`	Refers to record field of a given name
Index	`#index`	Refers to list element at a given index
Any	`*`	Refers to all children of a list node

Kind *arguments*	
List *tag*, $index_1$, $child_1$, . . ., $index_n$, $child_n$	Ordered list of nodes, addressable by *index*. Renders as <tag> with children.
Record *tag*, $field_1$, $child_1$, . . ., $field_n$, $child_n$	Record with children addressable by *field*. Renders as <tag> with children.
Reference *selectors*	Reference to another document location. Displays the `selectors` as a link.
Primitivie *string* or *number*	Numerical or textual primitive value. Renders as an HTML text node.

Figure 2: Structure of selectors and document nodes

Document Edits. The supported document edits and their behavior are listed in Fig. 3. All edits require *target* to which they are applied. Target is an absolute reference not containing the Parent selector. It can contain the Any selector, in which case the edit is applied to multiple nodes simultaneously. Most edits can only be applied to target node(s) of a specified kind. As discussed in §6, both fields of a record and list elements are ordered and edits that add a new item take the index or field name of a previous node.

The edits allow any transformation of a document through a series of steps whose effect can be tracked by the substrate. As we saw, Denicek updates references when document structure changes. Fig. 3 distinguishes between edits that keep references in a document unchanged (above) and edits that affect references (below).

Renaming a field or wrapping a node converts any references into that location (§4.2). When deleting a field to which there is a reference in the document, Denicek rejects the edit. A copy edit can also be rejected, because it is ambiguous whether references referring to the original location should continue referring to the source or should be changed to point to the target of the copying (a reference cannot refer to two locations). why not have distinct Copy and Move operations to resolve this?

Automatic Reference Update. There are two situations in which automatic update of references is undesirable. If an edit is applied to a singular element of a list (reference contains the Index selector), references that refer into the list should be unchanged. Although such edits violate the invariant that collections are homogeneous, they typically do so only temporarily, for example during construction of a new list item (checking such edits is discussed in §6.1). Wouldn't this leave dangling references? Wouldn't it be better to evolve the references as these edits occur? Insertion into a list can default the contents based on the type to avoid such a transient type error.

Edit *arguments*	Target
Add *target, field, after, node* Add *node* as a *field* to the specified record *after* a given field.	Record
Append *target, index, after, node* Append *node* to the end of the specified list *after* a given field.	List
Reorder *target, permutation* Reorder items of a specified list according to a *permutation*.	List
DeleteItem *target, index* Delete the item at a given *index* of a specified list.	List
UpdateTag *target, tag* Change the tag of a specified list or record from to a new *tag*.	List or Record
PrimitiveEdit *target, transform* Apply primitive *transform* to the specified primitive.	Primitive
RenameField *target, old field, new field* Rename the field of a specified record from *old* to *new*.	Record
DeleteField *target, field* Delete the field *field* of a specified record.	Record
WrapRecord *target, tag, field* Wrap the specified node as a *field* of a new record with *tag*.	Any
WrapList *target, tag, index* Wrap the specified node as a sole element of a new list with *tag*.	Any
Copy *target, selectors* Copy nodes(s) from *selectors*, replacing the specified target(s).	Any

Figure 3: Summary of document edit types in Denicek

Documents can also contain values that can be of multiple different kinds (i.e., a union type). This is surprsing since there has been no prior mention of union types, and lists are homogeneous. In such cases, references should not be updated when the kind of the value changes. For example, a formula may be either unevaluated or evaluated. As discussed in §5.4, evaluation involves wrapping the formula node, but this should not affect references to the formula. To support those cases, edits that normally affect selectors (Fig. 3) have a *reference behavior* field that can be set to disable automatic reference updating. This annotation is required when the target contains the *Index* selector. I think it would be desireable to avoid all these complications. References should be automatically maintained so they "just work". I believe that evaluation should not change the type/shape of the document. The space for evaluation results should be be preallocated in some sense, either as extra structure in a formula, or as metadata attached to the tree. That said, I've tried various alternative approaches to this in Subtext without ever settling on a preferred solution.

Edits are "static". An important principle is that the effect an edit has on references inside the document does not depend on the current value of the document, only its type. This makes it possible to define merging solely in terms of edits, without reference to current document state. (The effect Copy has on the document structure depends on the existing document structure, but not on its value).

This design choice makes it impossible to encode computational logic directly in the edits (e.g., through conditional edits). As we will see in §5.3, such logic can be provided as an additional mechanism on top of the underlying Denicek substrate.

4.2 Primitive Operations

Denicek provides three primitive operations. A sequence of edits can be applied to a document, two sequences of edits can be merged and also checked for conflicts. Denicek identifies edit histories by a (git-like) hash, computed from the hash of the parent and the current edit. The hash is used to identify a common shared part of the history during conflict resolution and merging. what exactly does parent mean here? History prefix?

Applying Edits. When applying an edit, Denicek locates the target node and transforms it according to the edit. If the edit can affect references in the document (Fig 3, below), Denicek updates matching references in the document according to the rules shown in Fig. 4 provided that the reference behavior of the edit is not set to disable the updating (this is required when the target reference contains the Index selector). Reference update behavior for WrapRecord and WrapList differs in that the updated references as a result of WrapList use the All selector. Although WrapList specifies the index to be used for the newly created list item, we assume that the operation introduces a homogeneous list and the new reference should point to all eventual list items.

Also note that affected references in the document may be more specific than the edit target. For example, if we rename `old` to `new` at `/foo/*`, a reference `/foo/3/old` will become `/foo/3/new`. (A reference in the document cannot be more general, because such edit would have to contain the Index selector and this would require setting reference behavior to not trigger reference update.) Finally, if the document contains a reference that would be invalidated by the Copy or DeleteField edit, the edit is rejected.

Merging Edit Histories. Merging edit histories is used when two users edit document independently, but also when replaying edits in programming by demonstration. The operation works on two edit histories, E, E_1 and E, E_2, that have a shared prefix E. The merging operation is akin to git rebase. It turns edits E_2 into edits E_2' that can be reapplied on top of the other edit history, resulting in a new history E, E_1, E_2'. Note that the operation is not symmetric. If there are conflicts among the edits in E_1 and E_2, the result of E, E_1, E_2' will differ from the result of E, E_2, E_1'. We return to this problem in the next section when discussing conflict detection.

It might help to define a function signature for merging to make this clearer: $\mathcal{M}(P, L, R) = M$ or $\mathcal{M}_P(L, R) = M$. Merging is often thought of as symmetric, perhaps choose a name that is clearly asymmetric, like rebase, translate, or project.

The key operation that enables such reconciliation takes two individual edits that occurred independently, e_1 and e_2, and produces a sequence of edits $e_2', e_2'', \ldots$ that can be applied after e_1 and have the effect of e_2, modified to respect the effects of e_1. There are two aspects of such reconciliation:

(1) *Apply to Newly Added.* If e_2 is adding new nodes to the document, but e_1 modified the document through a selector that would also affect the new nodes added by e_2, we need to apply the transformation represented by e_1 to the nodes newly added by e_2. This is done by generating an additional edit, to be applied after e_2, that is based on e_1 but targets only the newly added nodes (more details can be found in §B.1).

RenameField *target, old field, new field* – Replace Field for matching references.
`/target/old_field/nested` ⇒ `/target/new_field/nested`

WrapRecord *target, tag, field* – Insert extra Field selector after matching prefix.
`/target/nested` ⇒ `/target/field/nested`

WrapList *target, index, tag* – Insert extra All selector after matching prefix.
`/target/nested` ⇒ `/target/*/nested`

Figure 4: How document edits transform references

StructureEffect *target* – Affects fields or structure of the target node.
RenameField, DeleteField, WrapRecord, WrapList, Copy

ValueEffect *target* – Transforms value, modifies list or adds an additional field.
Add, Append, Reorder, DeleteItem, PrimitiveEdit

TagEffect *target* – Modifies the tag of the target node.
UpdateTag

Figure 5: Effects of individual edit operations

(2) *Transform Matching References.* If e_2 targets a node that is inside a node whose structure is changed by e_1, the target reference in e_2 is updated in a way that corresponds to the new structure. This is done using the rules shown in Fig. 4, that apply when transforming references inside a document, although we also support the case when $e1$ is Copy (see §B.2 for details).

I believe that in general a "backwards" form of merge is also needed, as in my *retract* and the OT *Exclusion Transform*. A regular merge takes forking edits and composes them. The backwards form converts composed edits into forking edits (and can be combined with a forward merge to recompose in the opposite order). This is needed for cherry picking and reverts, however you don't seem to need that for this paper.

Related question: how do you merge an Append(target, index, after, node) with DeleteItem(target, after)? A famous counterexample in OT was that two insertions adjacent to a deletion could flip their location if the edits were reordered. I'll dig up the details of this counterexample for you. Some systems handle this by leaving behind a "tombstone" on delete to remember the position of the deleted item. I have a novel solution but I'm probably going to go back to tombstones anyway because they are simpler and handy in the diff UI.

To illustrate merging, consider a case where we created a list of work items `/todo`. In one branch, we add an additional item to the list (e_2). In another branch, we wrap the list in an extra <div> element (e_1) and add a checkbox to each work item (e_1'):

e_2 = **Append**(`/todo`, #1, #0, Do some work)

e_1 = **WrapRecord**(`/todo`, <div>, items)
e_1' = **Add**(`/todo/items/*`, done, **nil**, ☐)

If we want to append the edit e_2 after edits e_1, e_1', we need to update its target to reflect the additional wrapping (2) and we need to create and additional Add operation that will add the checkbox to the new item (1). The result is two edits e_2', e_2'':

e_2' = **Append**(`/todo/items`, #1, #0, Do some work)
e_2'' = **Add**(`/todo/items/#1`, done, **nil**, ☐)

If we performed the merge operation in the other order, we would simply append e_1, e_1' after e_2. Although this is not the case in general, the result will be the same regardless of the order here.

Conflict Resolution. When merging two sequences of edits, E, E_1 and E, E_2, it is sometimes possible to construct edit histories E_1' and E_2' such that the results of applying E, E_1, E_2' and E, E_2, E_1' are the same. However, this is not always the case. Two edits can conflict if they both modify the same value, if they transfrom the structure of a node in incompatible ways or if one modifies a node nested inside a node replaced or deleted by the other. Detected conflicts can be reported to the user, but we also use conflict detection to implement incremental evaluation (§5.5). Our treatment of conflicts necessarily differs in this respect from replication algorithms like Operational Transformation [?] and Convergent Replicated DataTypes [?] where conflicts are arbitrarily but deterministically resolved.

The Denicek substrate implements a conflict detection mechanism inspired by effect systems [64]. The mechanism is simple and tractable, but over-approximates conflicts, i.e. it may report a conflict even if two edits can be merged successfully (see §6.1). An effects describe how an edit affects the document structure and we distinguish between three types of effects as shown in Fig. 5.

We say that a set of effects F_1 conflicts with another set of effects F_2 if there are effects $f_1 \in F_1$ and $f_2 \in F_2$ such that they are of the same kind and the target of f_1 is a prefix of the target of f_2 or vice versa (allowing specific Index to match against All in any direction).

Given two edit histories E, E_1 and E, E_2, Denicek can use conflict detection to remove all conflicting edits from E_2 and produce a sequence of remaining edits $e_2', e_2'', \ldots$ that can be added after E, E_1 and do not conflict with edits in E_1.

To do this, we iterate over edits e_2 from E_2 and check if the dependencies of e_2 conflict with effects of (1) any of the effect e_1 from E_1 or (2) effects of any of the previously removed edits. Here, the dependencies of e_2 include its target, but also source of Copy and additional dependencies recorded explicitly as discussed in §5.5. If a conflict is detected, the edit e_2 is removed and its effect is recorded, so that we remove any subsequent edits that would depend on the removed edit.

The effect-style conflict detection is a nice idea! Removing conflicts before merging is also an interesting idea I hadn't considered. That might allow me to simplify my transformation rules.

5 Programming Experiences Implementation

The key claim of this paper is that the Denicek computational substrate suppports a range of experiences that make programming more concrete, collaborative, and interactive. In particular, Denicek's ability to merge edits is a unifying mechanism that provides essential features across the range. We demonstrate this claim with the Webnicek web-based programming system, which uses Denicek to support:

(1) Local-first collaboration (§5.1)
(2) Programming By Demonstration (§5.2§5.3)
(3) Incremental recomputation (§5.4, §5.5)
(4) Schema change control (§5.6)
(5) End-user debugging via provenance tracking (§5.7)

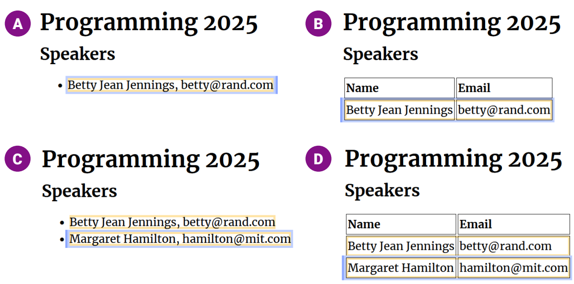

Figure 6: Merging of two independently done sequences of edits. Two ways of merging B and C result in the same D.

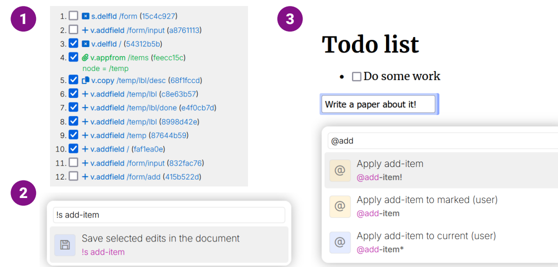

Figure 7: Programming by demonstration is implemented by selecting edits from the document history (1), saving them in the document (2) and replaying them (3).

(6) Concrete programming via managed copy & paste (§5.8).

We describe the programming experience in isolation in this section. The next section provides a more comprehensive evaluation through a case study that combines multiple of them together.

5.1 Local-First Collaboratation

The Denicek representation enables local-first collaboration [50], as illustrated previously in §3 (E). If a document is edited by multiple users, they can each make edits to their local copy and eventually merge the variants using the operation to merge edit histories.

Merging of histories behaves akin to git rebase in that it keeps a linear history. Synchronization in a distributed system thus requires first reapplying local edits on top of the remote history, before updating the remote history. Denicek thus implements the *convergence model* of document variants [25], i.e., the user cannot, for example, maintain their own local document structure and import new data from another variant (an alternative discussed in §6).

Recall that merging of edit histories is not symmetric. Document D in Fig. 6 can be obtained either by appending C' (produced by the edit reconciliation operation) on top of A, B or by appending B' on top of A, C. According to our effect analysis, the two operations are conflicting. Although B primarily affects the document structure, it also adds a new field to the record (email), which is a ValueEffect, conflicting with ValueEffect of C. In this scenario, the conflict can be ignored and the resulting document is the same in both cases.

However, the resulting two histories will differ. In the first case, the Append edit that adds a new node is supplemented by further focused edits that transform the added node from a list item to a new table row. In the second case, the structural transformations are automatically applied to all rows. As discussed in §4.2, conflicts during merging can be resolved either by removing conflicting edits or by letting the later edits overwrite the former ones.

To guarantee that replicas converge it is necessary to have a global order on merges. OT and CRDT algorithms use timestamps combined with a unique ID for each client to establish this global order without a centralized coordinator. We can do the same. This means you never know if your changes will dominate or whether they will be eventually overriden but people seem to be willing to live with it. Personally, I'm a skeptical: I expect that like with NoSQL people will eventually realize this sucks and go back to centralized systems.

5.2 Programming By Demonstration

In programming by demonstration [13], the user demonstrates a task to the system and the system then repeats it, directly or in a generalized way. To use direct repetition with Denicek (Fig. 7), the user can select edits from the edit history, name them and replay them. In case of general-purpose document editing in Webnicek, this requires certain forethought, but as illustrated in §7, the mechanism is very effective in a domain such as data wrangling [43].

There are two notable aspects of our implementation of programming by demonstration in Webnicek. First, Webnicek records edits in the document itself (by representing individual edits as nodes and storing them in list inside a `/saved-interactions` field). This means that no other implementation mechanism outside of the system is needed and also that the stored edits can be modified by the user or tools working with the document.

Second, to replay edits, Webnicek does not append the recorded edits on top of the current history. When saving edits, it stores the hash of the history at the time of saving. When replaying, it appends the recorded edits to the top of the original history (at the time of saving) and merges this new sequence of edits with the current history. This pushes the recorded edits through all subsequent edits made by the user. The result can be seen in Fig. 1 (E), where a newly added speaker is transformed from a list item to a table row.

Webnicek also accounts for the case where edits recorded in the document are themselves transformed (when they are reconciliated with other edits during merging). In this case, Webnicek updates the recorded edits (an alternative approach is discussed in §6).

Simply replaying recorded edits provides limited programming capabilities. We discuss in the next section how recorded edits can themselves be edited by the user to generalize and conditionalize their effects. More general and conventional programming constructs are discussed in §6.

5.3 Interactive User Interfaces

Programming by demonstration can be used to define interactive elements in the document. In a simple scenario, illustrated in Fig. 8 (1), the `click` event handler is set to a reference to a sequence of edits recorded in the document. Clicking the button executes the edits using the mechanism discussed in §5.2, i.e., Webnicek appends the edits to a history at the time when the edits were recorded and merges the edits with the current history.

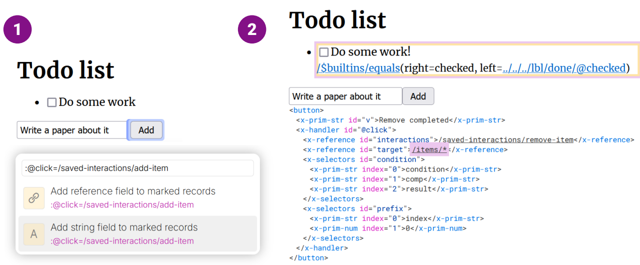

Figure 8: Using programming by demonstration to define a UI. The "Add" button (1) replays edits; the "Remove completed" button (2) modifies target and specifies a condition.

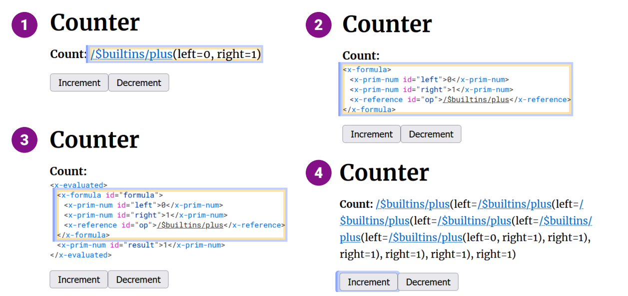

Figure 9: Increment wraps the existing count in a formula that adds 1 to the previous value (1), (2). Evaluation produces the count (3), which is invalidated on subsequent clicks (4).

The use of merging when replaying recorded edits is crucial in both the Todo App example and the Conference List example (see §A). In both cases, it makes it possible to define a user interface for adding items (new Todo items, new speakers) and later change the document structure (refactor speaker list to a table) or add functionality (formula to evaluate whether a Todo item has been completed) without having to recreate the user interface for adding items. The use of merging ensures that new items are added in a correct format or with the additional functionality.

Denicek can be used to generalize the interactions recorded through programming by demonstration. Our prototype illustrates this option in a limited way. As shown in Fig. 8 (2), a button to remove all completed Todo items can be created by generalising the `remove-item` interaction, which removes the list item at the index 0. In addition to the recorded interaction, we manually specify (in the source view) that the edits should be applied to all elements selected by the `/items/*` selector, instead of the original `/items/0` selector (prefix) and that the edits should only be applied to elements for which the formula (which tests if the checkbox is checked) specified by a relative selector `./condition/comp/result` evaluates to true.

The above paragraph was surprising to me and I'm not sure I fully understand it even after staring at the figure for a while. Edits are recorded as an AST using x-tags to hide them in the UI? That became clear only after reading the next section. Our audience might be a lot happier if we used a textual grammar. Earlier it was said that edits can't be conditional but then a conditional predicate appears here. This is a special feature of button nodes? Or event handler nodes? This starts to resemble Angular and similar template languages (which I find hard to read).

Generalization Heuristic. Specifying generalization manually is cumbersome. Programming by demonstration systems typically implement heuristic for generalization [70] infers and suggests such generalizations. If integrated into a system based on Webnicek, such heuristic could automatically construct a formula based on positive and negative examples [57] (selected and deselected items).

5.4 Formula Language and Evaluation

Denicek documents can contain formulas inspired by the spreadsheet paradigm [71]. Formulas can specify richer computations than what can be expressed using document edits. Formulas do not transform the document and their results are transient.

Formula evaluation leverages Denicek's ability to merge edit histories. As illustrated in Fig. 9, formulas are represented as document nodes with a special tag (`<x-formula>`). They are recognized by a formula evaluator and rendered in a special way (❶ and ❹), but they are created using ordinary edits and the Denicek substrate treats them as standard nodes.

To evaluate formulas, the formula evaluator generates edits that turn the `<x-formula>` record into `<x-evaluated>` (❷ and ❸), which keeps the previous formula state in the `formula` field and the evaluation result in the `result` field. (Keeping the previous state of the formula is not necessary, but it enables provenance analysis as discussed in §5.7.) The way edits generated by evaluation are merged with the document is discussed in the next section.

As I've said before, I'd prefer to store the result of the formula inside the <x-formula> record instead of wrapping it on evaluation. This avoids changing the shape of the document on evaluation, which I recall added some complexity earlier.

The Counter App example shown in Fig. 9 illustrates the interaction between formulas and programming by demonstration. To implement a counter, the Increment and Decrement buttons wrap the current counter value in a formula that adds or subtracts 1. I'd like to see the code for the increment button.

OK maybe I'm finally getting it. You aren't mutating the value of the counter, you're wrapping the value in an increment formula. I never got that in our previous discussions. Presumably that evaluation is now frozen so it serves as an execution trace. This is a lot different from my approach where a formula stays unchanged except for the results. One advantage is that fields contain a record of their computed mutations whereas in my approach that is recorded in the past state of the doc whence the mutation occured.

Formula Language. Webnicek exposes the underlying representation of formulas to the user, but the same representation can be edited through a user-friendly mechanism such as a block-based editor [39] or a calculation view [88]. The key point is that Denicek's tree structure makes it easy to embed formulas in document in a uniform way, edit them and merge them with other changes. Some readers may disagree that it is easy: AST editing is controversial.

5.5 Incremental Recomputation

As illustrated in Fig. 11, Webnicek supports incremental recomputation. In the example, the cost of speaker travel depends on the number of speakers, but the cost of refreshments depends only on

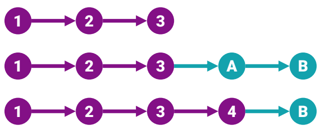

Figure 10: Evaluated edits (A, B) are kept at the top of the history. When they conflict with ordinary edits (4), they are removed (A).

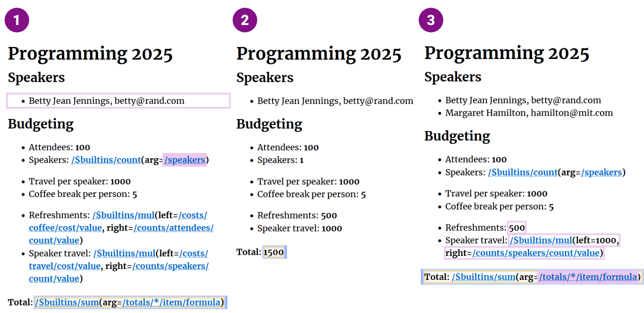

Figure 11: Budget calculation based on the number of speakers (1) and the result (2). When speaker is added (3), only the results of affected formulas are invalidated.

two constants. The edit that adds a speaker invalidates only the former computation.

The evaluation mechanism is illustrated in Fig. 10. This figure is out of order. When the formulas are evaluated, Webnicek generates *evaluated edits* that are appended to the top of the history. What exactly are these evaluated edits? When subsequent edits are made, they are appended after all non-evaluated edits and the evaluated edits are pushed through the newly added edits. If the edits conflict (according to conflict detection discussed in §4.2), affected evaluated edits and edits that depend on them are dropped.

Live Programming. The Webnicek prototype does not automatically evaluate formulas. This makes it easier to understand the evaluation mechanism, but a realistic system based on the substrate could automatically evaluate formulas to provide a live programming experience [79, 86] and use incremental recomputation for performance reasons.

I'm confused. Changing the value of a location conflicts with an edit that evaluates a formula containing a reference to that location? And these evaluation edits are second-class edits that get retroactively dropped to "unevaluate" the formula? I'd prefer what you suggest above: formulas are seemingly globally re-evaluated after every change, with incremental computation a hidden perf optimization.

An important property of incremental computation systems is avoiding redundant execution. Usually this involves a topological sort of the computation dependency graph. We may need to have a story about that to avoid hate from reviewer 2. Personally, I think incremental recomputation is about performance, not the programming experience, so it isn't that important for this paper.

5.6 Schema Change Control

Document structure often needs to evolve [10], as illustrated by our Conference List example where a list is transformed into a table. When this happens, data and code that depend on the structure

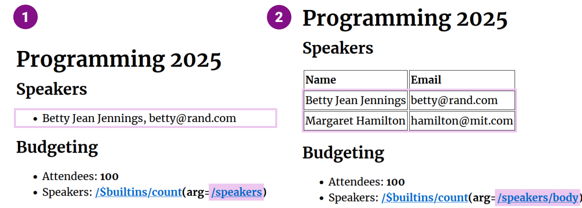

Figure 12: When an edit changes the document structure, references in formulas are updated accordingly.

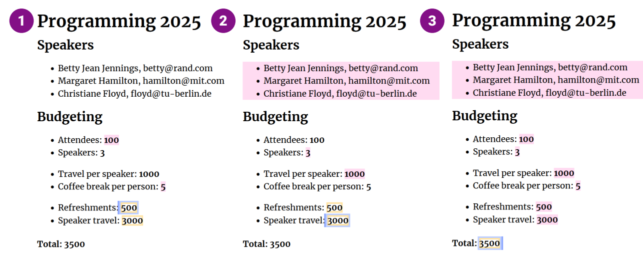

Figure 13: Using provenance tracking to highlight document parts that contributed to the calculation of refreshments costs (1), travel costs (2) and all costs (3).

of the document need to evolve correspondingly. The problem is well-known in database systems [83] and has recently been studied in the context of programming systems [25].

Although Denicek does not explicitly track document structure (schema or type), all documents have an implicit structure and some edits transform this structure. As discussed in §4.2, Denicek automatically updates reference nodes in the document when edits modify the document structure. This enables a form of schema code co-evolution [25]. Formulas embedded in Denicek documents use reference nodes to refer to both data sources (in the document) and the results of other computations. Consequently, if the document structure changes, the formulas are automatically updated.

Consider the example in Fig. 12. The original list `<ul>` is turned into `<tbody>` using UpdateTag and wrapped inside `<table>` with a field `body` using WrapRecord. For the latter, Denicek updates the reference accordingly turning the original `/speaker` reference in the formula into `/speaker/body`.

As noted earier, evaluated edits transform the formula structure (wrapping it in the `<x-evaluated>` record). However, they are marked as non-structural and so references to formulas are not transformed when the formula is evaluated (otherwise, references would be updated to point to the original unevaluated formulas).

5.7 End-User Debugging

The most common kind of end-user programming question is determining whether a value they observe is right or wrong [49]. One way to help users answer the question is to provide an explanation how a value was obtained [54]. Webnicek provides a basic mechanism that highlights document nodes that contributed to a specific computed result, illustrated in Figure 13.

The implementation leverages the fact that evaluation keeps the original formula, as discussed in §5.4. When a formula is evaluated, the final `<x-evaluated>` document node contains the result, but also sub-tree that represents a full evaluation trace [76]. We analyse the trace, collect all reference nodes in the trace and highlight all nodes referred to in the computation. Denicek makes this easy as we only need to collect reference nodes nested in a formula node.

Explanations and Linked Visualizations. In Webnicek, we implement provenance analysis to show inputs involved in computation, but information from the execution trace collected by the Denicek substrate can also be used to provide a detailed explanation [76] or to automatically generate linked visualizations [77].

5.8 Concrete Programming

Abstraction is an essential feature of programming, but it has a high cognitive cost [6]. Programming by demonstration (§5.2) offers one way of reducing the cost. Another way of making programming more concrete [20, 92] is to support copying of functionality as in prototype-based object-oriented programming [93].

Webnicek supports a functionality akin to managed copy & paste [22, 24] for formulas. Rather than introducing abstractions (functions), users can copy and modify formulas to reuse them. However, when the user discovers an error in the original formula, Webnicek lets them use the merging mechanism to correct the error in the original formula and all its copies.

The mechanism is shown in Figure 14. The user copies an incorrect formula using the Copy edit and then modifies it to use a different data source. They then navigate back in history to the point before the copying and create a temporary fork of the document. In the fork, they use RenameField to switch the argument of the division. When they merge the temporary fork into the original document, the *Apply to Newly Added* logic of the merge operation (§4.2) duplicates the RenameField edits and applies them also to the copied formula. In other words, the Copy edit of the Denicek substrate, alongside with the fact that formulas are ordinary document nodes, provide keys component for a straightforward implementation of the managed copy & paste functionality.

Nice and simple! My approach tried to be a lot fancier but ended up being too heavywieght.

Linked Editing. Webnicek currently requires users to explicitly manipulate history to correct error across multiple code clones. Research on managing duplicated code resulted in multiple tools [19, 97] with dedicated user interface to manage clones. The Denicek substrate provides the underlying mechanism that could be used to implement a more user-friendly interface inspired by those systems.

6 Design Considerations

Treat sequences of edits that need to be together as transactions?

The design of the Denicek substrate is the result of an iterative process in which we repeatedly adapted the Denicek design until we obtained a satisfactory solution for the six formative examples (§A) in the Webnicek prototype. In this section, we document the design challenges, many of which have until now been personal knowledge of researchers working on related systems [21, 31, 38, 73].

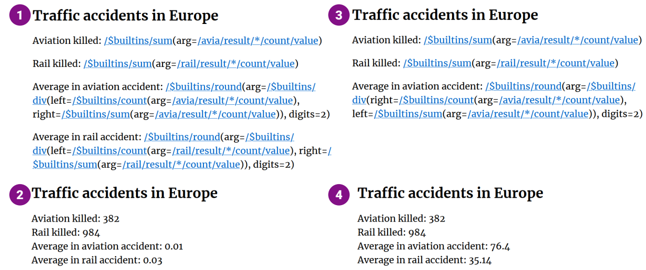

Figure 14: Correcting an incorrect formula. The user uses Copy to create the second formula (1). They notice an error (2), go back in history to switch left and right argument (3), merge the change and re-evaluate both formulas (4).

Two guiding principles for Denicek have been *composability* and *uniformity* [37]. The design should cover a large number of cases using a small number of concepts. This is apparent in the design of the document structure (a node can represent data, code or rich text) as well as in the role of edits (an edit can be a value or structure change, the result of user interaction or the result of evaluation).

The inconvenience associated with composability and uniformity [37] is not a problem for Denicek. We view Denicek as an underlying structure on top of which more convenient end-user programming systems can be built. The uniform design offers a degree of open-endedness, making it possible to use the basic structures of Denicek in new ways, not anticipated in this paper.

Implicit vs. Explicit Structure. A key design choice for a substrate like Denicek is whether to track the document structure explicitly or make it implicit. In Denicek, we assume that elements of a list node have the same structure (i.e. records should have the same fields, so that they can be targetted using the All selector), but we do not enforce this. Often, the property is temporarily violated when adding a list item, but then restored once the item is created.

Keeping the structure (type information) explicit has theoretical appeal and it simplifies some aspects of the implementation. In particular, edits to document structure can be clearly separated from edits of document values. The same separation has disadvantages in that it complicates the edit language and it requires user interface that departs further from document editor. (In Webnicek, users can select all list items and edit them at once using the same mechanism as when editing individual list items. With explicit structure, changing a list structure has to be either a different kind of edit or possibly an edit of a virtual "prototype" element.)

Merging and Document State. One specific design choice in Denicek is that the merge operation operates solely on two sequences of edits, but it does not need access to the current document state (or full edit histories). The reconciliation of edits (§4.2) is defined for two individual edits e_1 and e_2.

This design limits the possible language of edits. We cannot, in general, support conditional edits (that are applied only when a certain condition holds for the current document state). This is because the merging operation would not be able to determine whether the edit is applied (and whether its effects should take place).

One consequence of the design choice is that the generalization of recorded interactions (§5.3) has to be done on a meta-level (by specifying how to invoke edits, rather than through the edits themselves). Another consequence is that our langauge of selectors has to be abstract. We can target all list elements using the All selector, but cannot combine this representation with a representation that explicitly lists indices of all items currently in the list.

Explicit List Indices. Denicek does not use implicit numerical indices for lists. When adding an item, the user (or the system built on top of Denicek) has to explicitly provide an index. This design is motivated by a typical programming by demonstration scenario where the user adds a new list item and then modifies it (adding a speaker or a Todo item). When using explicit indices, the edits following Append can use the index to refer to and modify the newly added item. (To avoid overwriting existing items when replaying recorded edits, Webnicek replaces «««< HEAD indices of newly added items when replaying recorded interactions in §5.2). ======= indices of newly added items in recorded interactions in §5.2). I don't think that was explained back there. »»»> origin/review

Using implicit numerical indices is possible, but it requires treating all list-related edits as operations that transform document selectors. This means adding multiple new rules to Figure 4, which specifies how edits transform references. Merging then has to modify indices when edits add, remove or reorder list items. List order then, somewhat confusingly, becomes a part of document structure rather than a property of the list value.

«««< HEAD The problem could be avoided by not letting the user modify a list item after adding it. ======= I'm still unclear whether your list indices are ordinal numbers like in an array or are unique IDs. Your usage of implicit/explicit isn't obvious to me. The above paragraph seems to imply they are IDs. Note that automatically generating unique IDs in code execution can lead to nondeterminism. If you are using IDs, do they affect equality of lists?

The problem could also be avoided by not letting the user modify a list item after adding it. »»»> origin/review (We can require new items to be constructed outside of the list and then added through a single edit that combines the logic of Append and Copy.) This is easy to implement, but it requires unintuitive user interaction and it also does not address all use cases for modifying list item via index. I'm still confused, but this sounds like it really isn't a solution afterall.

Ordering List Items and Record Fields. When list indices are not numerical and supplied explicitly, we need another mechanism for tracking order of list items. Denicek uses a data structure inspired by Mergeable Replicated Data Types [42] and requires specifying the index of a preceding list item «««< HEAD when appending to a list. However, Webnicek supplies the index of preceding item automatically. (If two edits append to the same list independently with the same preceding item, the order is non-deterministic.) ======= when appending to a list. (If two edits append to the same list independently with the same preceding item, the order is non-deterministic.) Why is this needed? I let the order of operations determine the list order. »»»> origin/review

Note that Denicek uses the same mechanism for record fields. Keeping record fields ordered is desirable because Denicek documents map directly to HTML documents and so the order is visible to a user. Moreover, different ways of merging edits can result in different order of adding fields to a record. Making order explicit ensures those result in equivalent documents.

Denicek still makes a distinction between records and lists, even if the two are similar technically. The difference is conceptual. A list is expected to contain items of the same structure (that can be addressed using the All selector), whereas record fields are expected to be of different types. If we unified lists and records, All would have to be applicable to records too and inferring and checking document structure would be challenging.

Convergence vs. Divergence of Document Variants. There are two ways of managing document variants in collaborative editing [25]. In the *divergence* model, users can maintain their own variant indefinitely (share data, but use a different schema locally). In the *convergence* model, users have to adopt all earlier changes in order to adopt a later change (accept new schema before new data).

Denicek supports only the simpler *convergence* model. To support the *divergence* model, we would need an operation dual to our edit reconcilliation (given two subsequent edits, e_1, e_2, we want to generate e_2' that has the same effect as e_2 but can occur before e_1). The pair of operations has been called *project* and *retract* [23]. We choose not to support this as it would complicate the basic Denicek substrate structure.

Linear History vs. Graph of Edits. Denicek keeps a linear history of edits. When merging edits, new edits have to be transformed and added to the top of the history. This model is akin to git rebase. An alternative representation would be to maintain a graph of edits akin to when using git merge. In this representation, we would need to keep parents of edits and special merge edits would have multiple parents.

Maintaining a graph of edits would make recording of edits in programming by demonstration easier (§5.2) as we would not need to update recorded edits if they are transformed during merging. (Their hashes would remain the same.) However, supporting special merge edits and non-linear history would make the basic substrate more complex.

Supporting Rich Selectors. Denicek supports only a limited set of selectors. An absolute reference can select a record field (Field), specific list item (Index) or all list items (All). Once can imagine a range of other useful selectors, for example to select list items with a given tag or select items satisfying other conditions.

We could support such selectors for those edits that do not affect the document structure. However, supporting such conditional selectors in general is incompatible with our design decision that merging should not depend on document state. (If an edit wraps the body of all `<h3>` elements, we need to know the document state in order to determine whether we need to modify the selector pointing to an item at a specific index.)

Dependency Tracking. Recall that the incremental recomputation (§5.5) is based on the conflict detection mechanism of Denicek (§4.2). When pushing edit e_2 through e_1, a conflict is detected if they affect overlapping targets or if e_1 modifies a document node that is a source of e_2, typically, the source of a Copy edit.

For edits generated during formula evaluation (§5.4), we need to track an additional kind of dependency. Evaluating a formula that adds two numbers from two document location results in an edit that sets the result of the formula to the sum of the two numbers. The edit contains the resulting number, but we also need to record that the edit has the two source locations as dependencies. For this reason, Denicek makes it possible to annotate edits with additional dependencies. One alternative would be to introduce a special Evaluate edit that would identify an operation to compute and a list of arguments, specified as a list of references. We choose the former to keep the set of edits smaller.

6.1 Future work

Although Denicek does not explicitly track document structure (or schema, or type), all documents have an implicit structure. => use type system to check temporarily invalid state

better effect system

IDEA: Type check edit groups to ensure they preserve structure but not individual edits eg when adding list item

7 Case study

(data science environment)

what are the limitations of Datnicek

8 Evaluation

how well did Denicek work for implementing Datnicek

9 Discussion

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, November 2006.

[2] Marc T.P. Adam, Shirley Gregor, Alan Hevner, and Stefan Morana. Design science research modes in human-computer interaction projects. *AIS Transactions on Human-Computer Interaction*, 13(1):1–11, 2021.

[3] Michael D. Adams, Eric Griffis, Thomas J. Porter, Sundara Vishnu Satish, Eric Zhao, and Cyrus Omar. Grove: A bidirectionally typed collaborative structure editor calculus. *Proc. ACM Program. Lang.*, 9(POPL), January 2025.

[4] Manuel Bärenz. The essence of live coding: change the program, keep the state! In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2020, page 2–14, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. Structured editing for all: Deriving usable structured editors from grammars. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23. Association for Computing Machinery, 2023.

[6] A.F. Blackwell. First steps in programming: a rationale for attention investment models. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 2–10, 2002.

[7] Andrew Blinn, Xiang Li, June Hyung Kim, and Cyrus Omar. Statically contextualizing large language models with typed holes. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024.

[8] Marcel Borowski, Luke Murray, Rolf Bagge, Janus Bager Kristensen, Arvind Satyanarayan, and Clemens Nylandsted Klokmose. Varv: Reprogrammable interactive software as a declarative data structure. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Zouhaier Brahmia, Fabio Grandi, and Barbara Oliboni. A literature review on schema evolution in databases. *Computing Open*, 02(2430001):1–54, 2024.

[10] Margaret M. Burnett and Brad A. Myers. Future of end-user software engineering: beyond the silos. In *Future of Software Engineering Proceedings*, FOSE 2014, page 201–211. Association for Computing Machinery, 2014.

[11] Weihao Chen, Xiaoyu Liu, Jiacheng Zhang, Ian Iong Lam, Zhicheng Huang, Rui Dong, Xinyu Wang, and Tianyi Zhang. MIWA: mixed-initiative web automation for better user control and confidence. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software Technology, UIST 2023, San Francisco, CA, USA*, pages 75:1–75:15. ACM, 2023.

[12] James Cheney, Stephen Chong, Nate Foster, Margo Seltzer, and Stijn Vansummeren. Provenance: a future history. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 957–964, New York, NY, USA, 2009. Association for Computing Machinery.

[13] Allen Cypher and Daniel Conrad Halbert. *Watch what I do: programming by demonstration.* MIT press, 1993.

[14] Liangrun Da and Martin Kleppmann. Extending json crdts with move operations. In *Proceedings of the 11th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '24, page 8–14, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu. Generalizing operational transformation to the standard general markup language. In *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work*, CSCW '02, page 58–67, New York, NY, USA, 2002. Association for Computing Machinery.

[16] Andrea A. diSessa. Thematic chapter: Epistemology and systems design. In Andrea A. diSessa, Celia Hoyles, Richard Noss, and Laurie D. Edwards, editors, *Computers and Exploratory Learning*, pages 15–29, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[17] Andrea A. diSessa and Harold Abelson. Boxer: A reconstructible computational medium. *Commun. ACM*, 29(9):859–868, 1986.

[18] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–12. Association for Computing Machinery, 2020.

[19] Ekwa Duala-Ekoko and Martin P. Robillard. Clonetracker: tool support for code clone management. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 843–846, New York, NY, USA, 2008. Association for Computing Machinery.

[20] Jonathan Edwards. Example centric programming. *SIGPLAN Not.*, 39(12):84–91, December 2004.

[21] Jonathan Edwards. Subtext: uncovering the simplicity of programming. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 505–518, New York, NY, USA, 2005. Association for Computing Machinery.

[22] Jonathan Edwards. First class copy & paste. Technical Report MIT-CSAIL-TR-2006-037, Massachusetts Institute of Technology, 2006.

[23] Jonathan Edwards and Tomas Petricek. Typed image-based programming with structure editing, 2021. Presented at Human Aspects of Types and Reasoning Assistants (HATRA'21), Oct 19, 2021, Chicago, US.

[24] Jonathan Edwards and Tomas Petricek. Interaction vs. abstraction: Managed copy and paste. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments*, pages 11–19, 2022.

[25] Jonathan Edwards, Tomas Petricek, Tijs van der Storm, and Geoffrey Litt. Schema evolution in interactive programming systems. *The Art, Science, and Engineering of Programming*, 9(?):1–34, 2025.

[26] Richard P. Gabriel. The structure of a programming language revolution. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, page 195–214, New York, NY, USA, 2012. Association for Computing Machinery.

[27] Camille Gobert and Michel Beaudouin-Lafon. Lorgnette: Creating malleable code projections. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, UIST '23, New York, NY, USA, 2023. Association for Computing Machinery.

[28] Max Goldman, Greg Little, and Robert C. Miller. Real-time collaborative coding in a web ide. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, page 155–164, New York, NY, USA, 2011. Association for Computing Machinery.

[29] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, August 2012.

[30] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, August 2012.

[31] Christopher Hall, Trevor Standley, and Tobias Hollerer. Infra: structure all the way down: structured data as a visual programming language. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, page 180–197, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. Deuce: a lightweight user interface for structured editing. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 654–664, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Felienne Hermans and Tijs Van Der Storm. Copy-paste tracking: Fixing spreadsheets without breaking them. In Alex McLean, Thor Magnusson, Kia Ng, Shelly Knotts, and Joanne Armitage, editors, *Proceedings of the First International Conference on Live Coding*, page 300. ICSRiM, University of Leeds, 2015.

[34] Joshua Horowitz and Jeffrey Heer. Engraft: An api for live, rich, and composable programming. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, UIST '23, New York, NY, USA, 2023. Association for Computing Machinery.

[35] Joshua Horowitz and Jeffrey Heer. Live, rich, and composable: Qualities for programming beyond static text. In *Proceedings of the 13th Annual Workshop on the Intersection of HCI and PL (PLATEAU 2023)*, 2023.

[36] Patricia Jablonski and Daqing Hou. Cren: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology EXchange*, eclipse '07, page 16–20, New York, NY, USA, 2007. Association for Computing Machinery.

[37] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. Technical dimensions of programming systems. *The Art, Science, and Engineering of Programming*, 7(13):1–59, 2023.

[38] Joel Jakubovic and Tomas Petricek. Ascending the ladder to self-sustainability: Achieving open evolution in an interactive graphical system. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2022, page 240–258, New York, NY, USA, 2022. Association for Computing Machinery.

[39] Bas Jansen and Felienne Hermans. Xlblocks: a block-based formula editor for spreadsheet formulas. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 55–63, 2019.

[40] Nima Joharizadeh, Advait Sarkar, Andrew D. Gordon, and Jack Williams. Gridlets: Reusing spreadsheet grids. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI EA '20, page 1–7, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Tim Jungnickel and Tobias Herb. Simultaneous editing of json objects via operational transformation. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, page 812–815, New York, NY, USA, 2016. Association for Computing Machinery.

[42] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[43] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, page 3363–3372, New York, NY, USA, 2011. Association for Computing Machinery.

[44] A. Kay and A. Goldberg. Personal dynamic media. *Computer*, 10(3):31–41, 1977.

[45] Alan C. Kay. The early history of smalltalk. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, page 69–95. Association for Computing Machinery, 1993.

[46] Stephen Kell. Unix, plan 9 and the lurking smalltalk. In *Reflections on Programming Systems: Historical and Philosophical Aspects*, pages 189–213. Springer, 2018.

[47] Stephen Kell and J. Ryan Stinnett. Source-level debugging of compiler-optimised code: Ill-posed, but not impossible. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '24, page 38–53. Association for Computing Machinery, 2024.

[48] Eugen Kiss. Comparison of object-oriented and functional programming for gui development. Master's thesis, Leibniz Universität Hannover, 2014.

[49] Cory Kissinger, Margaret Burnett, Simone Stumpf, Neeraja Subrahmaniyan, Laura Beckwith, Sherry Yang, and Mary Beth Rosson. Supporting end-user debugging: what do users want to know? In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '06, page 135–142. Association for Computing Machinery, 2006.

[50] Martin Kleppmann, Adam Wiggins, Peter Van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 154–178, 2019.

[51] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. Webstrates: Shareable dynamic media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, page 280–290, New York, NY, USA, 2015. Association for Computing Machinery.

[52] Clemens Nylandsted Klokmose, James R Eagan, and Peter van Hardenberg. Mywebstrates: Webstrates as local-first software. In *UIST'24: Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. ACM, 2024.

[53] Amy J Ko and Brad A Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158, 2004.

[54] Amy J. Ko and Brad A. Myers. Finding causes of program output with the java whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, page 1569–1578. Association for Computing Machinery, 2009.

[55] David Koop and Jay Patel. Dataflow notebooks: Encoding and tracking dependencies of cells. In *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*, Seattle, WA, June 2017. USENIX Association.

[56] Eva Krebs, Patrick Rein, Joana Bergsiek, Lina Urban, and Robert Hirschfeld. Probe log: Visualizing the control flow of babylonian programming. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming*, pages 61–67, 2023.

[57] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 542–553, New York, NY, USA, 2014. Association for Computing Machinery.

[58] Germán Leiva, Jens Emil Grønbæk, Clemens Nylandsted Klokmose, Cuong Nguyen, Rubaiat Habib Kazi, and Paul Asente. Rapido: Prototyping interactive ar experiences through programming by demonstration. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 626–637, 2021.

[59] Sorin Lerner. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–7. Association for Computing Machinery, 2020.

[60] Geoffrey Litt and Daniel Jackson. Wildcard: spreadsheet-driven customization of web applications. In *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming*, Programming '20, page 126–135, New York, NY, USA, 2020. Association for Computing Machinery.

[61] Geoffrey Litt, Daniel Jackson, Tyler Millis, and Jessica Quaye. End-user software customization by direct manipulation of tabular data. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2020, page 18–33, New York, NY, USA, 2020. Association for Computing Machinery.

[62] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. Peritext: A crdt for collaborative rich text editing. *Proc. ACM Hum.-Comput. Interact.*, 6(CSCW2), November 2022.

[63] Geoffrey Litt, Peter van Hardenberg, and Henry Orion. Project cambria: Translate your data with lenses. https://www.inkandswitch.com/cambria.html, 2020. Accessed: 2020-10-01.

[64] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 47–57. Association for Computing Machinery, 1988.

[65] Sean McDirmid. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 53–62. Association for Computing Machinery, 2013.

[66] Andrew M Mcnutt, Chenglong Wang, Robert A Deline, and Steven M. Drucker. On the design of ai-powered code assistants for notebooks. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA, 2023. Association for Computing Machinery.

[67] Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, page 161–174, USA, 2001. USENIX Association.

[68] Brad A. Myers, Amy J. Ko, and Margaret M. Burnett. Invited research overview: end-user programming. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '06, page 75–80. Association for Computing Machinery, 2006.

[69] Brad A. Myers, Richard McDaniel, and David Wolber. Programming by example: intelligence in demonstrational interfaces. *Commun. ACM*, 43(3):82–89, March 2000.

[70] Brad A. Myers, Richard McDaniel, and David Wolber. Programming by example: intelligence in demonstrational interfaces. *Commun. ACM*, 43(3):82–89, March 2000.

[71] Bonnie A. Nardi and James R. Miller. The spreadsheet interface: A basis for end user programming. In *Proceedings of the IFIP TC13 Third Interational Conference on Human-Computer Interaction*, INTERACT '90, page 977–983, NLD, 1990. North-Holland Publishing Co.

[72] Bonnie A. Nardi, James R. Miller, and David J. Wright. Collaborative, programmable intelligent agents. *Commun. ACM*, 41(3):96–104, March 1998.

[73] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. Filling typed holes with live guis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 511–525, New York, NY, USA, 2021. Association for Computing Machinery.

[74] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 86–99, New York, NY, USA, 2017. Association for Computing Machinery.

[75] Addy Osmani, Sindre Sorhus, Pascal Hartig, and Stephen Sawchuk. Todomvc: Helping you select an MV* framework. https://todomvc.com/, 2024. Accessed: 2024-12-12.

[76] Roly Perera, Umut A Acar, James Cheney, and Paul Blain Levy. Functional programs that explain their work. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 365–376, 2012.

[77] Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. Linked visualisations via galois dependencies. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022.

[78] Tomas Petricek. Foundations of a live data exploration environment. *The Art, Science, and Engineering of Programming*, 4(8):1–37, 2020.

[79] Tomas Petricek. Foundations of a live data exploration environment. *The Art, Science, and Engineering of Programming*, 4(8):1–37, 2020.

[80] Tomas Petricek, Gerrit J. J. van den Burg, Alfredo Nazábal, Taha Ceritli, Ernesto Jiménez-Ruiz, and Christopher K. I. Williams. Ai assistants: A framework for semi-automated data wrangling. *IEEE Trans. on Knowl. and Data Eng.*, 35(9):9295–9306, September 2023.

[81] Tomas Petricek, James Geddes, and Charles Sutton. Wrattler: Reproducible, live and polyglot notebooks. In *10th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2018)*, London, July 2018. USENIX Association.

[82] Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. Codestrates: Literate computing with webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, page 715–725, New York, NY, USA, 2017. Association for Computing Machinery.

[83] Erhard Rahm and Philip A. Bernstein. An online bibliography on schema evolution. *SIGMOD Rec.*, 35(4):30–31, December 2006.

[84] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. Babylonian-style programming. *The Art, Science, and Engineering of Programming*, 3(3):9–1, 2019.

[85] Trygve M H Reenskaug. User-oriented descriptions of smalltalk systems. *BYTE*, 6(8):146–151, August 1981.

[86] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. Exploratory and live, programming and coding: A literature study comparing perspectives on liveness. *The Art, Science, and Engineering of Programming*, 3(1):1–33, 2019.

[87] Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. Imperative functional programs that explain their work. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–28, 2017.

[88] Advait Sarkar, Andrew D. Gordon, Simon Peyton Jones, and Neil Toronto. Calculation view: multiple-representation editing in spreadsheets. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 85–93, 2018.

[89] Jonas Schürmann and Bernhard Steffen. Lazy merging: From a potential of universes to a universe of potentials. In *11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation - Doctoral Symposium*, volume 82. Electronic Communications of the EASST, 2022.

[90] Nischal Shrestha, Titus Barik, and Chris Parnin. Unravel: A fluent code explorer for data wrangling. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, page 198–207, New York, NY, USA, 2021. Association for Computing Machinery.

[91] Jeremy Singer. Notes on notebooks: is jupyter the bringer of jollity? In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2020, page 180–186, New York, NY, USA, 2020. Association for Computing Machinery.

[92] David Canfield Smith. *Pygmalion: A Creative Programming Environment.* PhD thesis, Stanford University, 1975. Available as Stanford AI Memo AIM-260 and Computer Science Report STAN-CS-75-499.

[93] Randall B. Smith, John Maloney, and David Ungar. The self-4.0 user interface: manifesting a system-wide vision of concreteness, uniformity, and flexibility. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '95, page 47–60, New York, NY, USA, 1995. Association for Computing Machinery.

[94] Guy L. Steele and Richard P. Gabriel. The evolution of lisp. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, page 231–270. Association for Computing Machinery, 1993.

[95] Xin Tan, Xinyue Lv, Jing Jiang, and Li Zhang. Understanding real-time collaborative programming: A study of visual studio live share. *ACM Trans. Softw. Eng. Methodol.*, 33(4), April 2024.

[96] Tim Teitelbaum and Thomas Reps. The cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, September 1981.

[97] M. Toomim, A. Begel, and S.L. Graham. Managing duplicated code with linked editing. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 173–180, 2004.

[98] David Ungar and Randall B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242, December 1987.

[99] Tijs van der Storm. Semantic deltas for live dsl environments. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 35–38, 2013.

[100] April Wang, Zihan Wu, Christopher Brooks, and Steve Oney. Don't step on my toes: Resolving editing conflicts in real-time collaboration in computational notebooks. In *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments*, IDE '24, page 47–52, New York, NY, USA, 2024. Association for Computing Machinery.

[101] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 286–300, New York, NY, USA, 2019. Association for Computing Machinery.

[102] Pierre Étienne Meunier. Version control post-git. Presented at FOSDEM 2024, 2024. Accessed: 2025-03-24.

A Formative Examples

The design the Denicek substrate, we identified six formative examples shown in Fig. 15. The examples range from established industry benchmarks (Todo and Counter apps) to cases from literature [24] and problems posed as schema change challenges [25]. The Denicek substrate then co-evolved with Webnicek, a simple web-based programming environment built (as directly as possible) on top of the substrate and was used to solve implement the formative examples.

Many of the formative examples include a small programming challenge, such as adding user interface to add a new speaker, a new list item or modify the count. Our aim was for the substrate to enable solving those through programming by demonstration. Programming by Demonstration is often used in data wrangling [18, 30, 57]. Our Hello World example is only a minimalistic illustration of such use, loosely inspired by earlier work [67].

B Merging Edit Histories

Recall that merging takes two edit histories, E, E_1 and E, E_2, transforms edits E_2 into E_2' that can be reapplied on top of the first history resulting in E, E_1, E_2'. The key operation takes two individual edits, e_1 and e_2 and produces a sequence of edits $e_2', e_2'', \ldots$ that can be applied after e_1, and combine the two edits. This section provides details about the two aspects of this operation.

B.1 Apply to Newly Added

Assume that edits e_1 and e_2 occurred independently. We want to modify e_2 so that it can be placed after e_1. If the edit e_2 added new nodes to the document that the edit e_1 would affect, we generate an additional edit that apply the transformation of e_1 to the newly added nodes (and only to those).

The only edits that add new document nodes are Add, Append, Copy and so we consider this case if the edit e_2 is one of those. If so, we check whether the target of e_1 is within the target of e_2, i.e., the list of selectors that forms the target of e_2 is a prefix of the list of selectors that forms the target of e_1.

Along the way, we compute a *more specific prefix*. If the target of e_1 contains the All selector, it can be matched against a specific Index selector in the target of e_2 (if the selector of e_1 is more specific than that of e_2, the targets are not matched). We then replace the original prefix in e_1 with the *more specific prefix* that contains Index selector in places where the original edit contained All. This way, we obtain e_1' which is a focused version of e_1 that applies only to the nodes newly added by e_2. The edit e_2 thus becomes a pair of edits e_2, e_1'. The final document will contain edits e_1, e_2, e_1' – that is, it will first apply the edit e_1 to nodes already in the document, then add new nodes and then apply the transformation represented by e_1 to the newly added nodes.

B.2 Transform Matching References

As above, assume that edits e_1 and e_2 occurred independently. We want to modify e_2 so that it can be placed after e_1. If e_1 is any of the three edits listed in Fig. 4 (RenameField, WrapRecord, WrapList), we collect all references that appear inside e_2 (the target, the source of Copy and any references occurring in the nodes added by Add or Append). If the target of e_1 is a prefix of any of those references, we update the references accordingly and obtain a new edit e_2'. Note

Counter App [48] – Counter with increment and decrement buttons.
The current count is represented by a formula that is modified by the buttons. The user can inspect the evaluation trace to see how the count was modified.
Programming by Demonstration, Incremental Recomputation, End-User Debugging

Todo App [75] – Buttons to add an item and remove all completed.
Adding an item must correctly merge with independently added functionality to compute which items are completed and remove them based on a formula result.
Programming by Demonstration, Local-First Collaboration, Incremental Recomputation

Conference List [25] – Manage a list of invited conference speakers.
Adding speakers to a list through an in-document user interface merges with refactoring that turns the list into a table and separates name from an email.
Local-First Collaboration, Programming by Demonstration

Conference Budget [25] – Calculate budget based on a speaker list.
References are updated when the list is refactored. Only affected formulas are recomputed and the user can view elements on which the result depends.
Local-First Collaboration, Incremental Recomputation, End-User Debugging

Hello World [67] – Normalize the capitalization of two word messages.
An operation to normalize the text in a list item can be recorded and applied to all list items or, alternatively, applied directly to all list items.
Programming by Demonstration

Traffic Accidents [24] – Compute statistics using two data sources.
Formula to compute statistics can be reused with a different data source; error correction is propagated automatically to the copied version of the formula.
Concrete Programming, Incremental Recomputation

Figure 15: Formative examples used in Denicek design

that it would be an error to match specific Index in e_1 with more general All in e_2, but this cannot happen – reference updating is not done when the target of e_1 contains Index.

Now consider the case when e_1 is Copy and the edit e_2 targets a node that is the source node of the copy operation (or any of its children). In this case, it is reasonable to require that the edit e_2 is applied to both the source and the target of the copy. (This is required by the refactoring done in the Conference Budget example.) We handle the case by creating a copy of e_2 with transformed selectors (target and, if e_2 is also Copy, also its source). To transform the selectors, we replace the prefix formed by the source of the Copy by a new prefix, formed by the target target of the Copy. We then add the new operation as e_2' if at least one of its selectors was transformed (typically target, but possibly also source).

NICE THINGS IN DATNICEK * merging of code edits (e.g. change parameter merges with adding a call to chain) * remove cell to which there are refs - does the susbtrate catch this?