

# 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, 17 pages. <https://doi.org/XXXX.XXXX>

## 1 Introduction

A computational substrate defines the structures with which programs are constructed, how the program state is represented and how it evolves [39]. 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 [46], while representing programs as lists enabled Lisp to become a language laboratory [104].

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 [93] that are difficult to implement for C/C++ [47, 48].

*Programming Experiences.* In this paper, we describe a programming substrate that makes it easy to develop diverse programming experiences [72]. Those include:

- *Local-First Collaboration.* Multiple users should be able to concurrently modify a single document and automatically merge their changes, preferably without a central server [52, 64].
- *Programming by Demonstration.* Allow users to construct simple programs by enacting the steps of the expected behavior using concrete examples [13, 60].
- *Incremental Recomputation.* When a part of a document changes, formulas whose result depends on it are invalidated and, possibly, automatically recomputed [35, 67, 107].
- *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, 65].
- *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 [56].
- *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, 44]. 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.
- To evaluate how Denicek simplifies the development of programming systems, we build an innovative data science notebook Datnicek (§7) and assess its implementation complexity (§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 [72], notational freedom and self-sustainability [38], liveness, interactivity and richness [36, 95]. We see programming more as interacting with a medium or a substrate [45, 53] than writing text [27]. We focus on systems that allow gradual progression from a user to a developer, as envisioned in Smalltalk [94] 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, 32, 39] 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 [45]. Webstrates [53] is a substrate based on synchronization of documents (but without edit histories) that has been use as the basis for multiple programming systems [8, 91].

*Edit Histories and Merging.* Manipulating programs through semantically meaningful edits is a technique used by structure editors [5, 33, 107]. The language of edits has been captured formally as an *edit calculus* [78] and edits have also been used as the basis of live programming environments [110].

Merging of edits is most frequently done in version control systems such as git. Systems based on a more rigorous design such as Pijul [113] delay merging to a later point by using a graph or a lattice [98]. 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 (OT) approach [15], where edit conflicts are reconciled, or conflict-free representation (CRDTs) [52, 64, 99]. In both approaches, supporting complex edits on tree structures remains a challenge [14, 42]. Mergeable replicated data types (MRDTs) [43] can merge updates provided there is a suitable relational representation of the data. Recent work on MySubstrates and Grove [3, 54] has been based on CRDTs, whereas Edwards et al. [23] use OT.

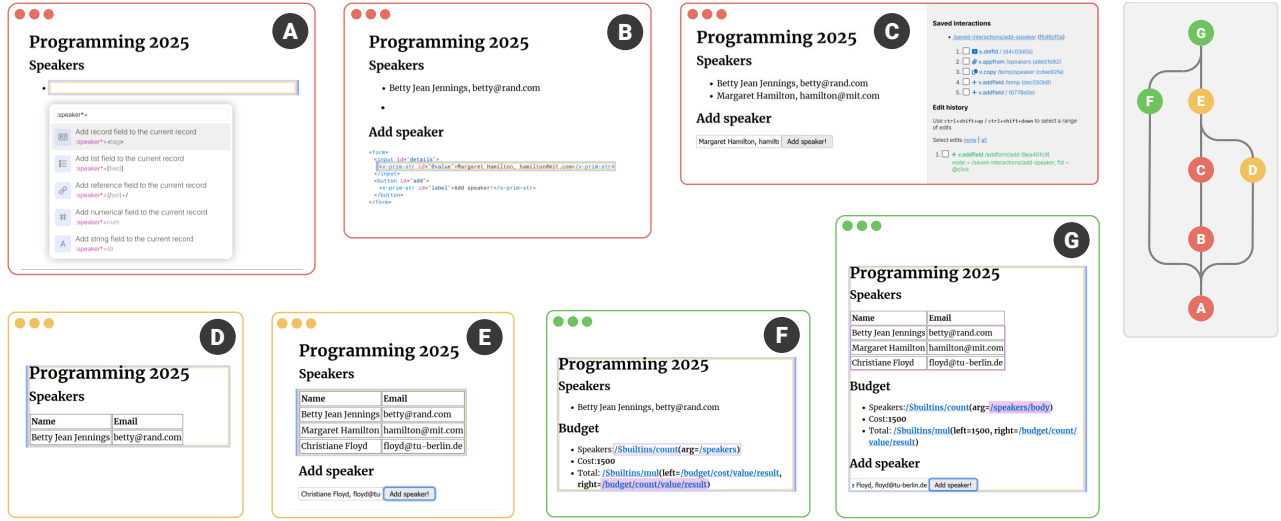
### 2.2 Programming Experiences

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 [29], real-time collaboration has become widely used, if not always without challenges [106]. Merging concurrent edits is one such challenge. In addition to CRDT-based approaches [54], conflicts arising during collaboration have been solved through fine-grained locking [111].

*Programming by Demonstration.* Earliest PbD systems used the paradigm for tasks ranging from graphics and user interfaces to general-purpose programming [13, 102]. Wrangler [44] showed the effectivity of PbD for data cleaning, whereas more recent uses range from augmented reality prototyping [60] and web automation [11] to end-user software customizations [62, 63]. Nevertheless, expressing conditions in PbD remains an active research topic [89, 90]. A more general class of demonstrational interfaces [73] also includes programming by example, used for example to infer data transformations in spreadsheets [30]. 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 [67, 83] 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 [107]. 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 [101]. This can be addressed by maintaining a dependency graph [57, 88] 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 [112]. However, the problem is starting to be recognized in programming systems research [25, 65] 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 [51]. Systems such as Whyline and Probe Log [55, 56, 58] 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 [100]. More generally, such functionality can leverage provenance tracking [12] and program slicing [80, 96]. The same infrastructure can also be used to build linked visualizations [81].

**Concrete Programming.** Programming can be simplified by working with concrete values instead of abstractions, an idea pioneered by the prototype-based programming language Self [109]. 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 feature [37, 108]. Notably, copy & paste has been also tracked in spreadsheets [34] and Gridlets [41] 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 [28], live literals [77], projection boxes [61] and data detectors [76] allow visualization or editing of aspects of programs through a user interface. The problem of making live and rich editors compositional is addressed by Engraff [35]. Programming environments may also integrate live examples [93] and a variety of AI assistance tools [7, 69, 87].

### 3 Walkthrough

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

**A Adding a Speaker.** The first user starts with an empty document, which is represented as a record. They add a field for each heading and a field named `speakers` containing a list `<ul>`. 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 a 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 interaction as an event handler for the `click` event of the button.



**Programming by Demonstration.** Denicek implements programming by demonstration (§5.2) by enabling the user to save and replay past interactions. Those can be replayed directly, or used as event handlers in order to construct interactive user interfaces (§5.3).

**D Refactoring Document Structure.** Another user starts with the initial version of the document (A) 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 a 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.** Formulas are document nodes and evaluating a formula yields additional edits that augment the document with the result (§5.4). Those evaluated edits are kept at the top of the document history and are removed in case of a conflict (§5.5), resulting in the incremental recomputation behavior.

**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 as discussed in §5.

### 4.1 Selectors, Documents and Edits

A computational document is a tree, consisting of four kinds of nodes (Fig. 2). Records and lists are labeled with a *tag* as in HTML. Record fields have unique names. Lists are expected to contain elements of the same structure, identified by a unique index (serving as an ID). Primitive nodes can be strings, numbers or references to another location in the document tree. References can be relative or absolute. Edit operations only use absolute references (to denote a target node), but relative references can appear in the document (e.g., to refer to another node within a list element).

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 elements of a list, but there is no way to refer to all fields of a

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




Kind	arguments
 <b>List</b>	<i>tag, index<sub>1</sub>, child<sub>1</sub>, ..., index<sub>n</sub>, child<sub>n</sub></i> Ordered list of nodes, addressable by <i>index</i> . Renders as <tag> with children.
 <b>Record</b>	<i>tag, field<sub>1</sub>, child<sub>1</sub>, ..., field<sub>n</sub>, child<sub>n</sub></i> Record with children addressable by <i>field</i> . Renders as <tag> with children.
 <b>Reference</b>	<i>selectors</i> Reference to another document location. Displays the selectors as a link.
<b>A</b> <b>Primitive</b>	<i>string or number</i> Numerical or textual primitive value. Renders as an HTML text node.

Figure 2: Structure of selectors and document nodes

record. Note that Denicek does not use implicit numerical indices for lists. The index of a new list item has to be supplied explicitly. The reasoning behind this design choice is discussed in §6.

**Document Edits.** The supported document edits and their behavior are listed in Fig. 3. All edits require a *target* to which they are applied. Targets are absolute references not containing the Parent selector. They 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, list elements as well as fields of a record are ordered and edits that add a new item take the index or field name of a previous node.

The edits allow an arbitrary transformation of a document via 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 updates any references to within the target location (§4.2). When deleting a field to which there is a reference in the document, Denicek rejects the edit. A copy edit of a node to which there is a reference is also rejected, because it is ambiguous whether references referring to the original location should be left unchanged, or modified to point to the target of the copying (a reference cannot refer to two unrelated locations).

**Automatic Reference Update.** As discussed in §6, the structure of Denicek document is implicit and not statically enforced (the substrate can be seen as dynamically typed). As a consequence, edits that transform the structure can be used as structural edits (affecting the structure) or as value edits (affecting the value of specific nodes). The latter can be used, for example, when constructing an additional list item. An item can be first constructed and then populated with values. This temporarily violates the invariant that collections are homogeneous (an issue we might address by using transactions as discussed in §6). Another use of value edits is when evaluating a formula, which involves wrapping the formula node (§5.4), but the operation should not affect references to the formula itself. To support value edits, edits that normally transform references (Fig. 3 below) have a *reference behavior* option that can be set



Edit	arguments	Target
<b>+</b>	<b>Add</b> <i>target, field, after, node</i> Add <i>node</i> as a <i>field</i> to the specified record <i>after</i> a given field.	Record
<b>@</b>	<b>Append</b> <i>target, index, after, node</i> Append <i>node</i> to the end of the specified list <i>after</i> a given field.	List
<b>↕</b>	<b>Reorder</b> <i>target, permutation</i> Reorder items of a specified list according to a <i>permutation</i> .	List
<b>⊖</b>	<b>DeleteItem</b> <i>target, index</i> Delete the item at a given <i>index</i> of a specified list.	List
<b>&lt;/&gt;</b>	<b>UpdateTag</b> <i>target, tag</i> Change the tag of a specified list or record from to a new <i>tag</i> .	List or Record
<b>I</b>	<b>PrimitiveEdit</b> <i>target, transform</i> Apply primitive <i>transform</i> to the specified primitive.	Primitive
<b>A</b>	<b>RenameField</b> <i>target, old field, new field</i> Rename the field of a specified record from <i>old</i> to <i>new</i> .	Record
<b>✖</b>	<b>DeleteField</b> <i>target, field</i> Delete the field <i>field</i> of a specified record.	Record
<b>📄</b>	<b>WrapRecord</b> <i>target, tag, field</i> Wrap the specified node as a <i>field</i> of a new record with <i>tag</i> .	Any
<b>≡</b>	<b>WrapList</b> <i>target, tag, index</i> Wrap the specified node as a sole element of a new list with <i>tag</i> .	Any
<b>📋</b>	<b>Copy</b> <i>target, selectors</i> Copy nodes(s) from <i>selectors</i> , replacing the specified target(s).	Any

Figure 3: Summary of document edit types in Denicek

to disable automatic reference updating. This annotation is required when the target contains the *Index* selector, which is necessarily a value edit affecting a specific node.

An important principle of the Denicek design 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. 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 they can be checked for conflicts. Denicek identifies edit histories by a (git-like) hash, computed from the current edit and the hash of the preceding edit. The hash is used to identify a common shared prefix of the history during conflict resolution and merging.

**Applying Edits.** When applying an edit, Denicek locates the target node and transforms it according to the edit. If the edit affects references in the document (Fig 3, below), Denicek updates the relevant references according to the rules shown in Fig. 4, provided that the reference behavior of the edit is not set to disable this.

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

**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

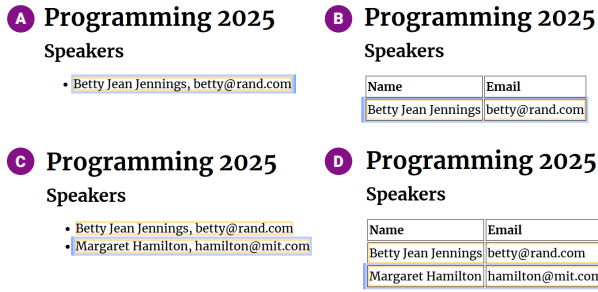
a homogeneous list, initially containing a single element, and the updated references should point to all eventual list items.

Note that the 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. (Contrariwise, a reference in the document cannot be more general. A more specific 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  $M_E(E_1, E_2)$  works on two edit histories,  $E, E_1$  and  $E, E_2$ , that have a shared prefix  $E$ . The 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. Formally,  $M_E(E_1, E_2) = E, E_1, E'_2$ . Note that the operation is not symmetric.  $M_E(E_2, E_1) = E, E_2, E'_1$  and the result of applying the two histories to the same node will differ if there are conflicts among the edits in  $E_1$  and  $E_2$ . We return to conflict detection in the next section.

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  $e'_2, e''_2, \dots$  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).
- (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 now also support the case when  $e_1$  is Copy (see §B.2 for details).



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

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 ( $e'_1$ ):

```
e2 = Append(/todo, #1, #0, <li>Do some work</li>)
e1 = WrapRecord(/todo, <div>, items)
e'_1 = Add(/todo/items/*, done, nil, <input type="checkbox"/>)
```

If we want to append the edit  $e_2$  after edits  $e_1, e'_1$ , we need to update its target to reflect the wrapping (2) and we need to create and additional Add operation that will add the checkbox to the new item (1). The result is a sequence with two additional edits  $e'_2, e''_2$ :

```
e1 = WrapRecord(/todo, <div>, items)
e'_1 = Add(/todo/items/*, done, nil, <input type="checkbox"/>)
e'_2 = Append(/todo/items, #1, #0, <li>Do some work</li>)
e''_2 = Add(/todo/items/#1, done, nil, <input type="checkbox"/>)
```

If we performed the merge operation in the other order, we would append  $e_1, e'_1$  after  $e_2$  without a change. In this case, the result of applying the two edit sequences would be the same.

**Conflict Resolution.** When merging two sequences of edits,  $E, E_1$  and  $E, E_2$ , it is desirable that applying  $M_E(E_1, E_2)$  and  $M_E(E_2, E_1)$  results in the same document. This is not always the case. If the two edits modify the same value, transform the structure of a node in incompatible ways or one edit modifies a node deleted by the other, the merge operation cannot reconcile. Such conflicting edits can be detected, reported to the user and optionally removed. Conflict detection is also used to implement incremental evaluation (§5.5).

The Denicek substrate implements a conflict detection mechanism inspired by effect systems [66]. 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). 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  $f_2 \in F_2$  that are of the same kind and the target of one is a prefix of the target of the other (allowing a specific Index to match against All in both directions).

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  $e'_2, e''_2, \dots$  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 the 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.

## 5 Programming Experiences

The key claim of this paper is that the Denicek computational substrate supports a range of compelling user experiences, many of which crucially rely on Denicek's ability to merge edits. We demonstrate this claim with the Webnicke web-based programming system, which uses Denicek to support:

- (i) local-first collaboration (§5.1),
- (ii) programming by demonstration (§5.2 and §5.3),
- (iii) incremental recomputation (§5.4 and §5.5),
- (iv) schema change control (§5.6),
- (v) end-user debugging via provenance tracking (§5.7),
- (vi) concrete programming via managed copy & paste (§5.8).

We describe the programming experience in isolation in the context of Webnicke in this section. The data science notebook system Datnicke, discussed in §7, provides a more comprehensive evaluation by combining multiple user experiences together.

### 5.1 Local-First Collaboration

The Denicek substrate enables local-first collaboration [52], as illustrated earlier (§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).

As an example, consider the scenario in Fig. 6. Here, document  $D$  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  $A, C$ . The edit histories resulting from the two ways of merging will differ. In the first case ( $B$  then  $C'$ ), the Append edit that adds a new node is followed by further focused edits that transform the newly added node from a list item to a table row. In the second case ( $C$  then  $B'$ ), the structural transformations are applied to all rows.

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 the ValueEffect of  $C$ . In this scenario, the conflict can be ignored and the resulting document will be the same. However, as discussed in §4.2, merging of edit histories is not symmetric and arising conflicts can also be resolved by removing conflicting edits.

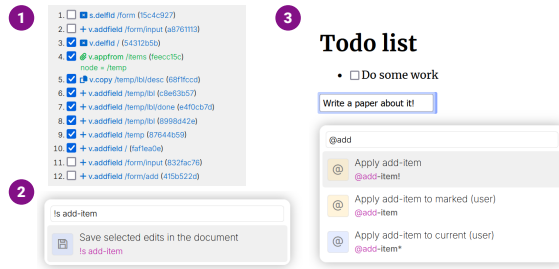


Figure 7: Programming by demonstration is implemented by selecting edits from the document history (1), saving them in the document (2) and replaying them (3).

## 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 selects edits from the edit history, name them and replay them. For 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 [44].

There are two notable aspects of our implementation of programming by demonstration in Webnicek. First, Webnicek records the saved edits in the document itself (by representing individual edits as nodes and storing them in a list inside the `/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 reconciled with other edits during merging). In this case, Webnicek updates the recorded edits (an alternative approach is 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.

The use of merging when replaying recorded edits is crucial in both the Todo App and the Conference List formative examples (see §A). In both cases, the merging makes it possible to define a user interface for adding items (new Todo items, new speakers) and later change the document structure (refactor the speaker list into a

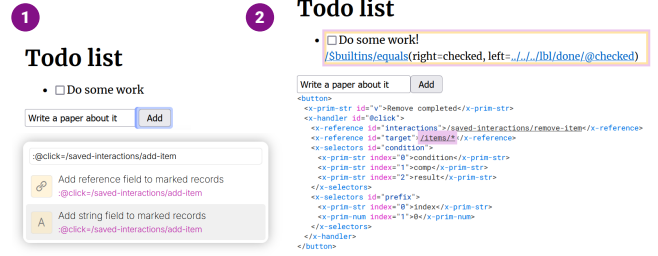


Figure 8: Using programming by demonstration to define a UI. The “Add” button (1) replays edits; the “Remove completed” button (2) modifies target and specifies a condition.

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.

In addition to the core functionality provided by Denicek, the Webnicek system also makes it possible to generalize the interactions recorded through programming by demonstration 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.

Note that the generalization mechanism does not violate the Denicek principle discussed in §4.1 that edits cannot depend on values. Webnicek finds all nodes for which the condition holds and generates one specific (non-conditional) edit for each of the nodes.

**Generalization Heuristic.** Specifying generalization manually is cumbersome. Programming by demonstration systems typically implement heuristic for generalization [74] infers and suggests such generalizations. If integrated into Webnicek, such heuristic could for example automatically construct a formula based on positive and negative examples [59] (selected and deselected items).

## 5.4 Formula Language and Evaluation

As illustrated earlier (§3, F), Denicek documents can contain formulas inspired by the spreadsheet paradigm [75]. Formulas in Webnicek do not transform the document and their results are transient, but they can describe richer computations than what can be expressed using document edits.

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 (4), 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>` (2) record into `<x-evaluated>` (3), which keeps the previous formula state in the `formula` field and the evaluation

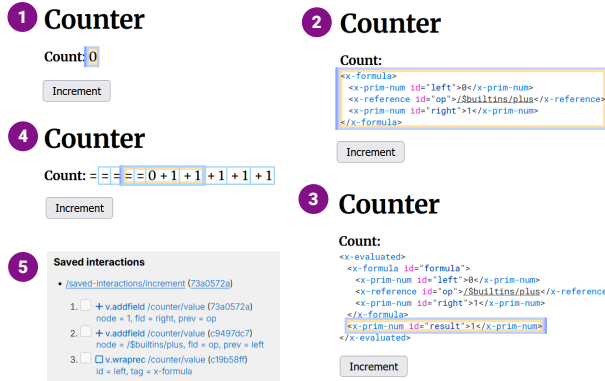


Figure 9: We wrap the initial count (1) in a formula that adds 1 to the value, (2). Evaluation produces the count (3), which is invalidated on subsequent clicks (4). The button replays the saved edits that wrap the current count in a formula (5).

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.

The Counter App example shown in Fig. 9 illustrates the interaction between formulas and programming by demonstration. To implement a counter, we first add the initial value 0 to the document. We then perform three edits that wrap the current counter value in a formula that adds 1 to the value. The three edits are used as an event handler for the button and so clicking the button creates an increasingly long sequence of increments. One advantage of this approach that we will leverage later is that the system also records a trace of how the final computed value was obtained.

**x<sup>2</sup> Formula Language.** Webnicke exposes the underlying representation of formulas to the user, but the same representation can be edited through a user-friendly mechanism such as a textual calculation view [97] a block-based editor [40]. The key point is that Denicek's tree structure makes it easy to embed formulas in document in a uniform way, edit them as any other node and merge them with other changes.

## 5.5 Incremental Recomputation

As illustrated in Fig. 10, Webnicke supports incremental recomputation. The cost of speaker travel depends on the number of speakers, but the cost of refreshments depends only on two constants. The edit that adds a speaker invalidates only the former computation.

The evaluation mechanism is illustrated in Fig. 11. When the formulas are evaluated, Denicek generates edits that wrap the formula in `<x-evaluated>` and set the result to the computed value. In Webnicke, those evaluated edits are kept in a separate list that is always appended to the top of the current edit history. When the user makes subsequent edits, the evaluated edits are pushed through the newly added edits. If the edits conflict (according to the conflict detection discussed in §4.2), the affected evaluated edits and edits that depend on them are dropped.

The dependencies between edits alongside with the conflict detection mechanism provide functionality that other systems implement using a dependency graph using topological sort.

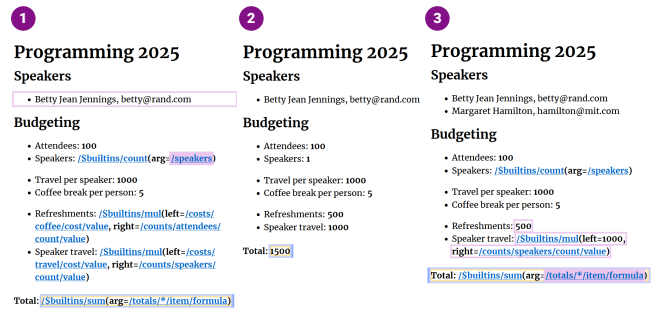


Figure 10: Budget calculation based on the number of speakers (1) and the result (2). When speaker is added (3), only the results of affected formulas are invalidated.

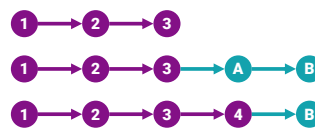


Figure 11: Evaluated edits (A, B) are kept at the top of the history. When they conflict with ordinary edits (4), they are removed (A).



**Live Programming.** The Webnicke 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 [84, 95] and use incremental recomputation for performance reasons.

## 5.6 Schema Change Control

Document structure often needs to evolve [10], as illustrated by the formative example Conference List where a list is transformed into a table. When this happens, data and code that depend on the structure of the document need to evolve correspondingly. The problem is well-known in database systems [92] and has recently been explored 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 and code co-evolution [25]. Formulas embedded in 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. 13. 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`.

Edits produced by document evaluation transform the formula structure (wrapping it in the `<x-evaluated>` record). However, the reference behaviour of those edits is set to keep references unchanged (as discussed in §4.1) 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).



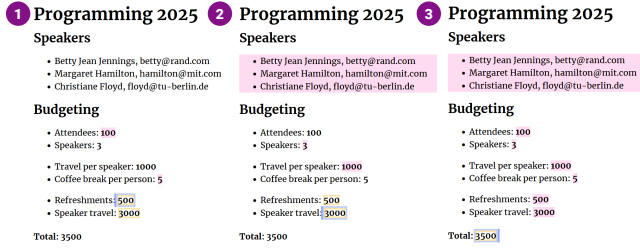


Figure 12: Using provenance tracking to highlight document parts that contributed to the calculation of refreshments costs (1), travel costs (2) and all costs (3).

## 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 [51]. One way to help users answer the question is to provide an explanation how a value was obtained [56]. Webnicek provides a basic mechanism that highlights document nodes that contributed to a specific computed result, illustrated in Figure 12.

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 the full evaluation trace [80]. 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 need to collect references directly nested in the 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 [80] or, for example, to automatically construct linked visualizations [81].

## 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, 102] is to support copying of functionality as in prototype-based object-oriented programming [103, 109].

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. 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.

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 also applies them to the copied formula. The merging operation with the Copy edit and the fact that formulas are ordinary document nodes, provide keys component for a straightforward implementation of the managed copy & paste functionality.

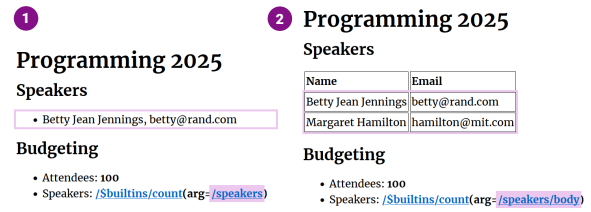


Figure 13: When an edit changes the document structure, references in formulas are updated accordingly.



**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, 108] 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

The design of the Denicek substrate is the result of an iterative process in which we repeatedly adapted the Denicek design and revisited the implementation of the Webnicek system until we obtained a satisfactory solution for the six formative examples (§A). In this section, we document the design challenges, many of which are shared with related systems [21, 32, 39, 77].

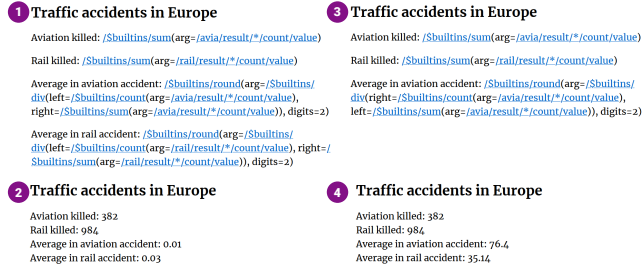


**Uniformity and Composability.** Two guiding principles for the design of Denicek have been *composability* and *uniformity* [38]. The substrate should cover a large number of scenarios using a small number of concepts. This is apparent in the design of the document structure—a node can represent data, 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.

**Dynamic and Static Typing.** Denicek does not explicitly track the document structure (it is dynamically typed). We assume that lists are homogeneous, but do not enforce this property. A statically typed system could distinguish structural edits from value edits. This has a theoretical appeal and it simplifies aspects of the implementation, e.g. by removing the need to control reference updating, but it makes working with the system less akin to document editing.

We violate structural invariants when constructing values gradually (e.g., when adding a new speaker to a table), as well as in evaluation (wrapping `<x-formula>` in `<x-evaluated>`). A system with static types could use a form of edit transactions, where structural invariants are reestablished only after a sequence of edits.

**Expressive Selectors and Edits.** Our merge operation operates on two sequences of edits, but it does not need access to the current document state or history. This limits the expressiveness of edits. Denicek does not support conditional edits (applied only when a specific condition holds), because merging would then not be able to determine if two edits are conflicting. As a result, generalization of recorded interactions (§5.3) has to be done at a meta level and generates a series of individual edits. This design also limits the expressiveness of selectors. For example, an edit targeting nodes with a specific tag (e.g., all `<h3>` elements) would also be conditional.



**Figure 14: The user uses creates a copy of a formula (1). They notice an error (2), go back in history to switch the arguments (3), merge the change and re-evaluate both formulas (4).**

*List Indices and Ordering.* Denicek does not use numerical indices for lists. Indices are unique identifiers that have to be provided explicitly. (Although Webnicek generates indices automatically when editing lists.) The design supports a programming by demonstration scenario where the user adds a new list item and then modifies it (adding a speaker or a Todo item).

With numerical indices, the edits following Append would not have access to the index of the newly added item. Using a numerical index computed from the list length would require complex logic to update indices when merging edit operations that affect lists. Another alternative is to disallow modification of a newly added item, but this limits programming by demonstration.

To maintain order of list elements, Denicek uses a data structure inspired by the list MRDT [43] and requires specifying the index of a preceding item when appending or inserting into a list. Denicek uses the same mechanism for record fields as the order of fields may be visible to the user, for example in Webnicek where fields represent children of a HTML node. Making the order of fields explicit ensures that it is maintained during merging.

*Merging Structure and Capabilities.* Denicek keeps a linear history and merging appends edits to the top of the history. This model is akin to git rebase. An alternative is to maintain a graph of edits akin to git merge. This would simplify recording of edits in programming by demonstration (§5.2) as we would not need to update saved edits 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.

Denicek also only supports the *convergence* model of collaborative editing [25] where users have to merge all changes in order and cannot adopt selected edits (cherry picking in git). In the *divergence* model, users could keep their own schema but import all data edits. Supporting the model requires a *retract* operation [23] that is dual to our edit reconciliation (given subsequent edits  $e_1, e_2$ , we want to generate  $e'_2$  that has the same effect as  $e_2$  but can occur before  $e_1$ ).

*Dependency Tracking.* Conflict detection in Denicek is used when merging edits, but also to implement incremental recomputation (§5.5). Edits generated during formula evaluation (§5.4) may be Copy edits (when evaluating a reference), but also Add edits (when setting the evaluation result to a computed value). For those edits, we record additional dependencies identifying the source of the operation arguments. Those additional dependencies also need to be considered in conflict detection.



**Figure 15: Underlying document representation of a code cell and a grid cell in the Datnicek notebook system.**

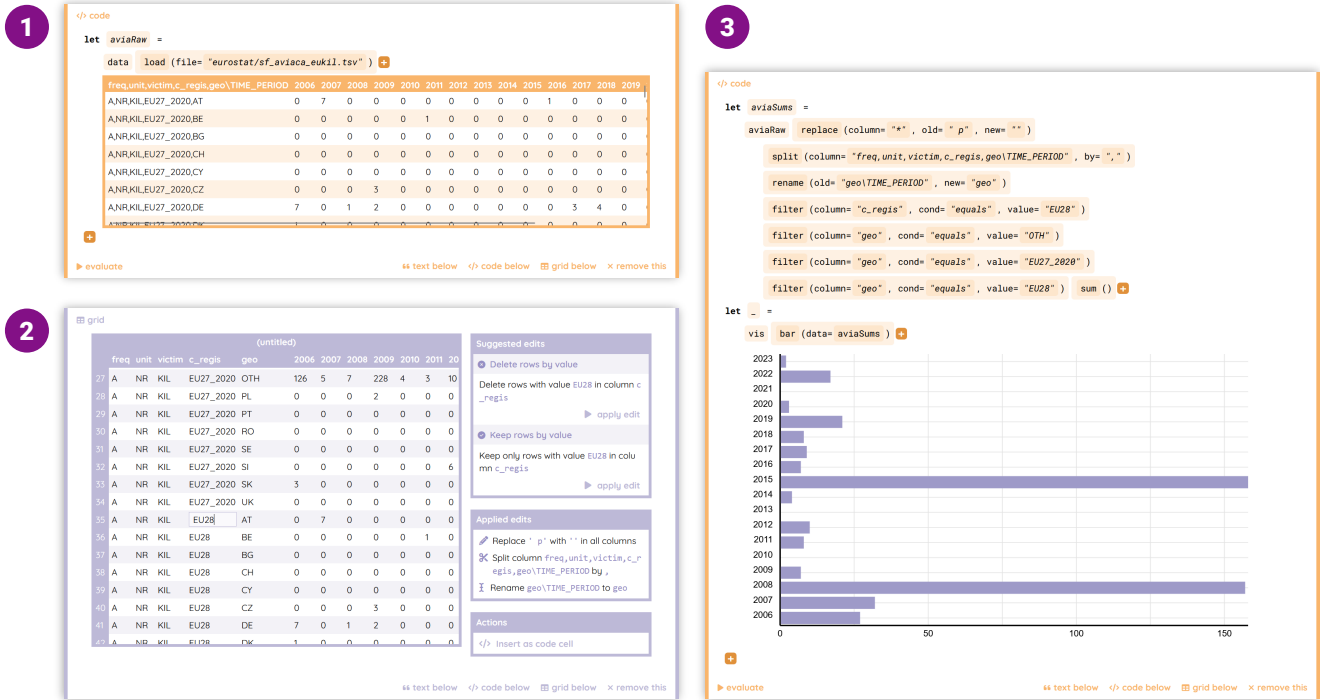
## 7 Case study: Datnicek Notebooks

Denicek is a low-level computational substrate. It serves best as the basis for interactive programming systems that view programs as documents. The most prominent example of such systems today are notebook environments for data science. To explore this use case, we have developed Datnicek, a simple notebook system that shows the ability of Denicek to support rich interactive user experiences. In this section, we present Datnicek and reflect on its development. A simple data exploration conducted in Datnicek is shown in Fig. 16.

### 7.1 Requirements

The design of Datnicek brings together a range of recent research ideas on interactive programming environments for data science. Datnicek notebooks consist of code cells and markdown cells, but they also support interactive grid cells where the user can use programming by demonstration to construct data cleaning scripts. We aim to support the following features:

- *Structure Editing.* Code in code cells should be edited through structure editor as in Histogram [82]. This allows Datnicek to keep track of the code structure and potentially makes the system accessible to non-programmers [68]. We do not yet aim to implement advanced keyboard-based editing [5, 71].
- *Collaborative Editing.* It should be possible to merge independently done code changes as in Grove [3], addressing the known pitfall of versioning Jupyter notebooks [101].
- *Code Completion.* During editing, code completion should offer available data transformations and operations as when using type providers [105] or iterative prompting in The Gamma [86].



**Figure 16: A notebook visualizing Air traffic accident data from Eurostat. The user loads data in a code cell (1), edits it in a grid that infers edit operations via programming by demonstration (2), turned to a code cell (3) and then visualized.**

- *Output Invalidation.* When code is edited, the previously evaluated results that depend on it, directly or transitively, should be invalidated as in Wrattler [83, 88], addressing another well-known limitation of Jupyter notebooks [57].
- *Interactive Data Cleaning.* It should be possible to edit data tables in an interactive grid and use programming by demonstration for common data cleaning tasks as in Wrangler or Vizier [44, 49].
- *Wrangling Code Synthesis.* Interactively constructed data transformations should be turned into code that can be checked by the data scientist and further edited as in Wrex [18].

The Datnicek notebook system implements the above requirements on top of Denicek. Although Datnicek is a proof of concept, it shows that Denicek provides a suitable basis for the implementation.

## 7.2 Implementation

Datnicek is implemented using the Elm architecture [26], where a system maintains a state that is updated through events. The state consists of the list of Denicek edits, transient state of the user interface, current computed document and other cached information. All interactions that affect the notebook use an event that appends edits to the list of Denicek edits. Remaining events update the transient user interface state. The support for collaborative editing has been implemented through an ad-hoc mechanism.

*User Interface.* Datnicek notebooks consists of a series of cells. In the code cells, the user can use the “+” button to add a new variable, add an operation to the end of an existing chain of operations, but also to specify an argument of an operation. The button opens a menu showing all valid options based on the current context.

In the interactive grid, the user can edit column headers and table cells. When they edit a value, Datnicek suggests generalised edits to be applied to the entire column or table. For example, when the user changes the “0 p” value to “0”, Datnicek suggests to replace “p” with the empty string in all cells. Other suggested edits include renaming or deleting a column, splitting a column using a delimiter and filtering rows based on the selected value. Datnicek can also turn edits performed interactively into a new code cell.

*Programmatic Code Cells.* A Denicek document representing a Datnicek notebook is shown in Fig. 15. A notebook is a record storing individual cells as fields. The tag determines the type of the cell.

The programming language used in code cells is inspired by the data exploration calculus [83]. A cell consists of a sequence of bindings of the form  $\text{let } v = e$  that assign an expression  $e$  to a variable  $v$ . An expression can be a reference to a variable, global value (such as `data` and `vis`) or a method invocation  $e.m(e_1, \dots, e_n)$ .

Bindings are represented as fields of a record and expressions use the formula representation discussed in §5.4. Operations and global values are identified by an absolute reference pointing to the special `/ $datnicek` namespace. Chains of method calls are represented using nested formulas. Evaluation of formulas proceeds as in Webnicek and wraps `<x=formula>` in `<x=evaluated>`. The user interface displays the original formula alongside with the evaluated result, supporting tables and Compost data visualizations [85]. Edits to the formula invalidate dependent results as discussed in §5.5.

*Interactive Grid Cells.* Grid cells consist of a reference to the data source they edit and a collection of `<transform>` nodes that represent individual data transformations constructed via programming

by demonstration. A transformation consists of metadata, a sequence of edits and a formula representing the transformation (with `<x-hole>` standing for the data table to be transformed).

A transformation may correspond to multiple underlying Denicek edits. For example, splitting “foo/bar” using “/” as the delimiter adds a new column “bar”, copies values from “foo/bar”, renames “foo/bar” to “foo” and then transforms the values in the new columns by dropping the part before and after the delimiter, respectively.

Interactive grid also supports conditional transformations that affect rows satisfying a given condition (e.g., deleting rows with a specific value in a given column). As discussed in §4.1, Denicek does not allow specifying conditional selectors. To support conditional transformations, we add a special kind of (virtual) edit `<x-expandable-edit>` that stores the underlying edit and a condition. As in §5.3, when applying the edit, Datnicek finds all rows for which the condition holds and generates a single non-conditional edit for each such row.

7.3 Reflections

edits and formula in interactive should be unified

foof  
(data science environment)  
what are the limitations of Datnicek  
x  
x  
x  
x

8 Evaluation

how well did Denicek work for implementing Datnicek

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?



## 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 Griffiths, 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 Bergsieck, 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 Symposium 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 Klokmoose. 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 Vansummen. 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] Simon Fowler. Model-view-update-communicate: Session types meet the elm architecture. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15–17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPICs*, pages 14:1–14:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- [27] 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.
- [28] 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.
- [29] 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.
- [30] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, August 2012.
- [31] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, August 2012.
- [32] 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.
- [33] 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.
- [34] 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.
- [35] Joshua Horowitz and Jeffrey Heer. Engraff: 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.
- [36] 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.
- [37] 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.
- [38] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. Technical dimensions of programming systems. *The Art, Science, and Engineering of Programming*, 7(13):1–59, 2023.
- [39] 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.
- [40] 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.
- [41] 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.
- [42] 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.
- [43] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.
- [44] 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.
- [45] A. Kay and A. Goldberg. Personal dynamic media. *Computer*, 10(3):31–41, 1977.
- [46] 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.
- [47] Stephen Kell. Unix, plan 9 and the lurking smalltalk. In *Reflections on Programming Systems: Historical and Philosophical Aspects*, pages 189–213. Springer, 2018.
- [48] 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.
- [49] Oliver Kennedy, Boris Glavic, Juliana Freire, and Mike Brachmann. The right tool for the job: Data-centric workflows in vizier. *IEEE Data Eng. Bull.*, 45(3):129–144, 2022.
- [50] Eugen Kiss. Comparison of object-oriented and functional programming for gui development. Master's thesis, Leibniz Universität Hannover, 2014.
- [51] 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.
- [52] 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.
- [53] Clemens N. Klokmoose, 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.
- [54] Clemens Nylandstedt Klokmoose, 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.
- [55] 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.
- [56] 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.
- [57] 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.
- [58] 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.
- [59] 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.
- [60] Germán Leiva, Jens Emil Grønbaek, Clemens Nylandstedt Klokmoose, 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.
- [61] 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.
- [62] 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.
- [63] Geoffrey Litt, Daniel Jackson, Tyler Millis, and Jessica Quayle. 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.
- [64] 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.

- [65] 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.
- [66] 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.
- [67] 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.
- [68] Andrew McNutt and Ravi Chugh. Projectional editors for json-based dsls. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 60–70, 2023.
- [69] 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.
- [70] 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.
- [71] David Moon, Andrew Blinn, and Cyrus Omar. tylr: a tiny tile-based structure editor. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2022, page 28–37, New York, NY, USA, 2022. Association for Computing Machinery.
- [72] 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.
- [73] Brad A. Myers, Richard McDaniel, and David Wolber. Programming by example: intelligence in demonstrational interfaces. *Commun. ACM*, 43(3):82–89, March 2000.
- [74] Brad A. Myers, Richard McDaniel, and David Wolber. Programming by example: intelligence in demonstrational interfaces. *Commun. ACM*, 43(3):82–89, March 2000.
- [75] Bonnie A. Nardi and James R. Miller. The spreadsheet interface: A basis for end user programming. In *Proceedings of the IFIP TC13 Third International Conference on Human-Computer Interaction*, INTERACT '90, page 977–983, NLD, 1990. North-Holland Publishing Co.
- [76] Bonnie A. Nardi, James R. Miller, and David J. Wright. Collaborative, programmable intelligent agents. *Commun. ACM*, 41(3):96–104, March 1998.
- [77] 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.
- [78] 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.
- [79] Addy Osmani, Sindre Sorhus, Pascal Hartig, and Stephen Sawchuk. Todomvc: Helping you select an MV\* framework. <https://todomvc.com/>, 2024. Accessed: 2024-12-12.
- [80] 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.
- [81] 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.
- [82] Tomas Petricek. Histogram: You have to know the past to understand the present. Presented at Workshop on Live Programming (LIVE'19), 2019.
- [83] Tomas Petricek. Foundations of a live data exploration environment. *The Art, Science, and Engineering of Programming*, 4(8):1–37, 2020.
- [84] Tomas Petricek. Foundations of a live data exploration environment. *The Art, Science, and Engineering of Programming*, 4(8):1–37, 2020.
- [85] Tomas Petricek. Composable data visualizations. *Journal of Functional Programming*, 31:e13, 2021.
- [86] Tomas Petricek. The gamma: Programmatic data exploration for non-programmers. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–7, 2022.
- [87] Tomas Petricek, Gerrit J. J. van den Burg, Alfredo Nazabal, 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.
- [88] 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.
- [89] Kevin Pu, Rainey Fu, Rui Dong, Xinyu Wang, Yan Chen, and Tovi Grossman. Semanticcon: Specifying content-based semantic conditions for web automation programs. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, UIST '22, New York, NY, USA, 2022. Association for Computing Machinery.
- [90] Marissa Radensky, Toby Jia-Jun Li, and Brad A. Myers. How end users express conditionals in programming by demonstration for mobile apps. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 311–312, 2018.
- [91] Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmoose. 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.
- [92] Erhard Rahm and Philip A. Bernstein. An online bibliography on schema evolution. *SIGMOD Rec.*, 35(4):30–31, December 2006.
- [93] 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.
- [94] Trygve M H Reenskaug. User-oriented descriptions of smalltalk systems. *BYTE*, 6(8):146–151, August 1981.
- [95] 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.
- [96] 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.
- [97] 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.
- [98] 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.
- [99] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [100] 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.
- [101] 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.
- [102] 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.
- [103] 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.
- [104] 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.
- [105] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. Themes in information-rich functional programming for internet-scale data sources. *DDFP '13*, page 1–4, New York, NY, USA, 2013. Association for Computing Machinery.
- [106] 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.
- [107] Tim Teitelbaum and Thomas Reps. The cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, September 1981.
- [108] 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.
- [109] David Ungar and Randall B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242, December 1987.
- [110] Tijs van der Storm. Semantic deltas for live dsl environments. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 35–38, 2013.
- [111] 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.
- [112] 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.
- [113] 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. 17. 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, 31, 59]. Our Hello World example is only a minimalistic illustration of such use, loosely inspired by earlier work [70].

## 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  $e'_2, e'_2, \dots$  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 [50]** – 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 [79]** – 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 [70]** – 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 17: 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).