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

Tomas Petricek
tomas@tomasp.net
Faculty of Mathematics and Physics, Charles University
Prague, Czech Republic

Jonathan Edwards
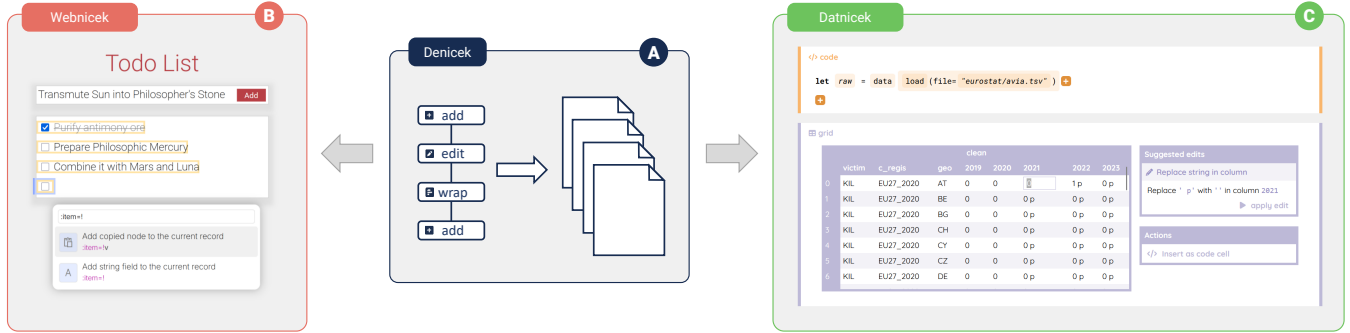jonathanmedwards@gmail.com
Independent
Boston, USA

Figure 1: Denicek is a computational substrate for document-oriented programming based on document edit histories (A). We co-design Denicek with a web-based end-user programming environment Denicek (B). Here, Denicek is used to build a Todo list app via programming-by-demonstration, by copying a value from input to a new list item. We then evaluate the generality of Denicek by using it to build a data science notebook system Datnicek (C). Here, Datnicek is used to interactively clean the data table, by removing the " p" (provisional) marker from numerical columns.

Abstract

User-centric programming research gave rise to a variety of compelling programming experiences, including collaborative source code 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. Denicek provides three operations on edit histories: edit application, merging of histories and conflict resolution. Many programming experiences can be easily implemented by composing these three operations.

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

1 Introduction

A computational substrate defines the structures with which programs are constructed, how the program state is represented and how the state evolves during execution [40]. The choice of a substrate affects what programming experiences can be readily supported. For example, object-oriented programming has been historically linked to graphical user interfaces [47], while representing programs as lists enabled Lisp to become a language laboratory [105].

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 [94] that are difficult to implement for C/C++ [48, 49].

Programming Experiences. We describe Denicek, a computational substrate that makes it easy to build programming systems supporting diverse compelling programming experiences [74]:

- *Collaborative Editing.* Users should be able to locally modify a shared document and merge their changes, preferably in a way that does not require a live connection to a central server [53].

- *Programming by Demonstration.* Allow users to construct simple programs by enacting the steps of the desired behavior using concrete examples and generalizing from the example [15, 61].

- *Incremental Recomputation.* When a part of a document changes, formulas whose result depends on the part are invalidated and, possibly automatically, recomputed [36, 69, 109].

- *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 [27, 67].

- *End-User Debugging.* The user should be able to ask provenance questions [13] to understand why a computation resulted in a particular value and what inputs contributed to the result [57].

- *Concrete Programming.* It should be possible to reuse parts of program logic, or formulas, without introducing abstractions, that is, program against concrete values [24, 26].

Two-Phase Methodology. The technical focus of this paper fits within the interior mode of design science research [2]. To *design* Denicek (Fig. 1 (A)), 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 a simple web-based end-user programming environment Webnicek (Fig. 1 (B)), which is 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* the usability of Denicek for the development of end-user programming systems, we use it to build Datnicek (Fig. 1 (C)), an innovative interactive data science notebook that brings together a range of recent research work [3, 20, 45, 87]. We reflect on the degree to which Denicek simplifies the implementation of a programming system that is based on current research advances and was conceived after the design of Denicek has been fully finalized (§7.3). We also provide a heuristic evaluation of Denicek characteristics including importance and generality (§8).

Computational Substrates. Denicek 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.

Key Takeaways. The key insights of this paper are twofold. A specific technical takeaway is that many compelling programming experiences can be implemented on top a uniform document representation, by using a suitable composition of three primitive operations that operate on sequences of edit operations.

Editing of data and formulas is done using a single set of primitive edit operations that manipulate the document. A user-interface may provide a specialized editor, but still trigger the primitive edits behind the scenes. 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 with 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, yielding an incremental recomputation behavior (§5.5).

A more basic takeaway, illustrated by the development of Denicek, is that it is worth looking for novel computational substrates that better support compelling programming experiences developed in user-centric programming research.

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 on top of 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 and assess its implementation complexity (§7). We also present heuristic evaluation of the system (§8).

To enable others build on top of Denicek, we share our compact open-source implementation at: https://github.com/d3sprog/denicek

2 Background

The premise of Denicek is that suitable structures for constructing programs can make it easier to support a range of compelling user experiences. A prime example is Smalltalk whose object-oriented structures enable malleable programming experience [14, 108].

Denicek aims to support a class of systems associated with end-user programming, liveness and interactivity [37, 74, 96], notational freedom and self-sustainability [39]. We see programming as interacting with a medium or a substrate [29, 46, 54] 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 follow systems such as BootstrapLab [40], which aims to support gradual progression from a user to a developer envisioned in Smalltalk [95] in a primarily graphical environment and Subtext [23], which develops a user-friendly programming environment based on object copying.

2.1 Programming Systems

Programming Systems and Substrates. A number of systems illustrate the qualities Denicek aims to support and have a related underlying structure. Subtext, BootstrapLab and Infra [23, 33, 40] 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 [19], which introduced the *naive realism* principle (what the user sees is all there is) that we also 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.* The term also dates back to Boxer [18] and is related to the notion of dynamic media of Kay and Goldberg [46]. Webstrates [54] revisit the idea, providing a substrate based on synchronization of documents (without edit histories) that has been used as the basis for multiple programming systems [8, 92].

Edit Histories and Merging. Manipulating programs through semantically meaningful edits is a technique used by structure editors [5, 34, 109]. The language of edits has been captured formally as an *edit calculus* [80] and edit histories have also been recognized as a suitable basis for live programming environments [112].

Merging of edits is most frequently done in version control systems. The Pijul system [115] delays merging to a later point by using a graph or a lattice [99]. In the context of programming environments, Grove [3] uses a commutative patches with a similar graph structure as the basis for a collaborative structure editor. Grove patches (construct, delete, relocate) are sufficient for a structure editor, but lack some structures and operations that our formative examples rely on (collections, copying). However, the core model of Grove provides a possible alternative basis for Denicek.

More generally, merging of edits can use the operational transform (OT) approach [17], where edit conflicts are reconciled, or conflict-free representation (CRDTs) [65, 100]. The latter is commonly associated with local-first software [53] that operates without a central server. In both approaches, supporting complex edits on tree structures remains a challenge [16, 43]. Mergeable replicated data types (MRDTs) [44] merge updates using a suitable relational representation. Recent work on MySubstrates and Grove [3, 55] has been based on CRDTs, whereas Edwards et al. [25] prefer OT.

Building Programming Experiences. The challenge addressed by Denicek is that end-user programming experiences are hard to build. Our experience developing such systems [22, 23, 40, 84, 87] suggests two difficulties. The first is the need to move between the concrete and the abstract [39]. A sequence edits strikes a balance between those opposing representations. The second is the need to combine multiple experiences, which requires a major engineering effort when using a conventional program representation.

2.2 Programming Experiences

Collaborative Editing. Since the early collaborative programming environments such as Collabode [31], real-time collaboration has became widely used, if not always without challenges [107]. Merging concurrent edits is one such challenge. In addition to OT-based and CRDT-based approaches [55], conflicts arising during collaboration have also been solved using fine-grained locking [113].

Programming by Demonstration. Earliest PbD systems used the paradigm for tasks ranging from graphics and user interfaces to general-purpose programming [15, 103]. Wrangler [45] showed the effectivity of PbD for data cleaning, whereas more recent uses range from augmented reality prototyping [61] and web automation [12] to end-user software customization [63, 64]. Expressing conditions in PbD remains an active research topic [90, 91]. A more general class of demonstrational interfaces [75] also includes programming by example, used for example for data transformation in spreadsheets [32]. Demonstrational interfaces can be used to directly perform actions, but also to generate code as in Wrex [20].

Incremental Recomputation. Interactive programming systems with live previews [69, 85] have been attempting to update the previews without full recomputation at least since the pioneering work on the Cornell Program Synthesizer [109]. Outside live programming, more work has been focused on updating computation when data change, although such data can also be source code passed to an adaptive interpreter [1]. Incremental recomputation is also a concern in notebook systems where cells can be run out of order [102]. The ordering problem can be addressed using a dependency graph [58, 89], allowing for 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 [114]. The problem is starting to be recognized in programming systems research [27, 67] 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 [52]. Systems such as Whyline and Probe Log [56, 57, 59] 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 [101]. More generally, such functionality can leverage provenance tracking [13] and program slicing [82, 97]. The same infrastructure can also be used to build linked visualizations [83].

Concrete Programming. Programming can be simplified by working with concrete values instead of abstractions, an idea pioneered by the prototype-based programming language Self [111]. In Self, prototypes are used at the object level. At the expression level, similar functionality can be provided by managed copy & paste. Subtext [24, 26] treats this mechanism as central, whereas other systems view tracking of copy & paste as an extra editor feature [38, 110]. Copy & paste has also been tracked in spreadsheets [35]. Gridlets [42] offer a spreadsheet abstraction based on reusing concrete computation, similar to the one proposed for The Gamma [85].

Other Programming Experiences. Several experiences not directly addressed in this paper are worth further investigation. Projectional editors such as Lorgnette [30], live literals [79], projection boxes [62] and data detectors [77] allow visualization or editing of aspects of programs through a user interface. The problem of making live and rich editors compositional is addressed by Engraft [36]. Programming environments may be also improved by integrating live examples [94] and a variety of AI assistance tools [7, 71, 88].
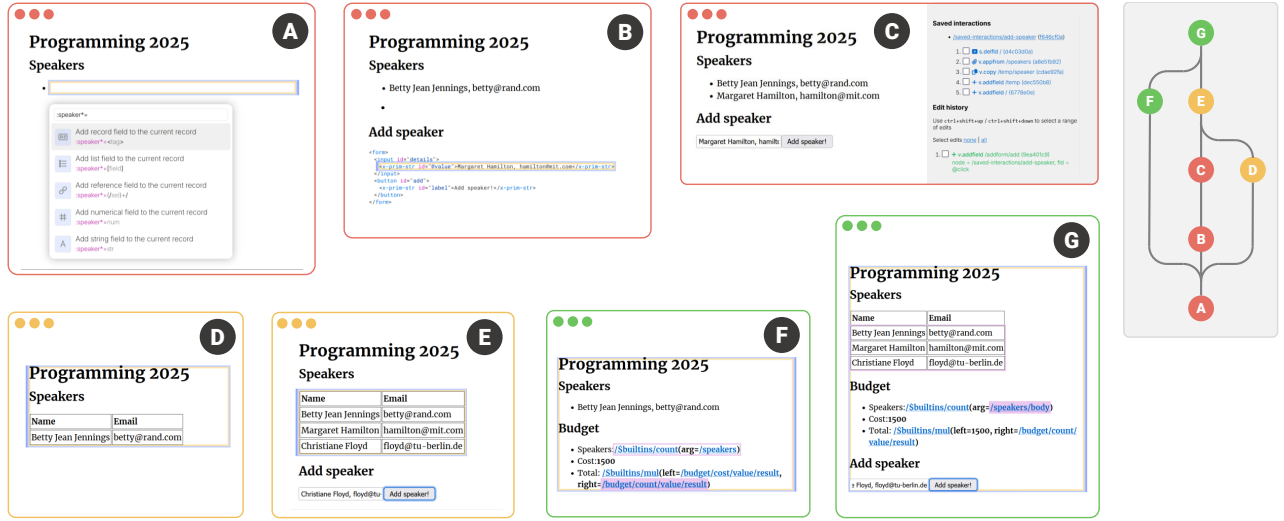
Figure 2: Organizing a conference using Denicek. The Walkthrough shows construction of a user interface for adding speakers (A, B, C); refactoring of the list and merging edits (D, E); and formulas with schema and code co-evolution (F, G).

3 Walkthrough

We use the web-based prototype Webnicek to illustrate programming experiences enabled by Denicek. Webnicek is based on a structure editor that supports navigation in the document and issuing of edit commands. We follow a formative example (§A) where Evelyn, Juliana and Alfred collaboratively plan a conference.

Ⓐ *Adding a Speaker.* Evelyn starts with an empty document, which is represented as a record. She adds a field for each heading and a field named speakers containing a list <ul>. She uses the command toolbox to issue edit commands that add the first speaker.

Ⓑ *Creating a User Interface.* To simplify adding of further speakers, Evelyn creates a textbox and a button using the command toolbox. She enters the details of another speaker into the textbox, adds a new <li> element and copies the speaker details from the textbox into the new element in the source view, using a copy command.

Ⓒ *Abstracting the Interaction.* After adding the speaker, Evelyn opens the history view, selects edits that added the speaker and saves them as the add-speaker interaction. She attaches 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. Past interactions can be replayed directly, or used as event handlers in order to construct interactive user interfaces (§5.3).

Ⓓ *Refactoring Document Structure.* Alfred starts with the initial version of the document (A) and turns the list into a table. He invokes a series of commands that rename tags, wrap elements, copy values and split strings using the comma as a separator.

Ⓔ *Merging Edits.* Alfred then merges the later edits (B, C) done by Evelyn into his version of the document. The refactoring is applied to all speakers. New speakers added using the "Add speaker!" button are also automatically transformed to 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).

Ⓕ *Adding Budget Calculation.* Juliana joins in and adds a budget calculation to the initial document. She uses the command toolbox to create document nodes representing formulas. Formulas are regular nodes with a special x-formula tag and arguments as child nodes. To count the speakers, she uses the count builtin with a reference to the node representing the speaker list as the argument.

> **Incremental Recomputation.** Formulas are document nodes. Evaluating a formula yields 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), a logic that implements incremental recomputation using core Denicek operations.

Ⓖ *Merging Formulas.* When Juliana merges the budget calculation (F) with the earlier edits (E), references in formulas are automatically updated to point to the rows of the table. Adding new speaker via the "Add speaker!" button invalidates results of only those formulas that depend on the number of speakers.

> **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.

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

	Kind *arguments*
☰	**List** *tag, index₁, child₁, ..., indexₙ, childₙ* — Ordered list of nodes, addressable by *index*. Renders as `<tag>` with children.
▢	**Record** *tag, field₁, child₁, ..., fieldₙ, childₙ* — Record with children addressable by *field*. Renders as `<tag>` with children.
↗	**Reference** *selectors* — Reference to another document location. Displays the `selectors` as a link.
A	**Primitive** *string* or *number* — Numerical or textual primitive value. Renders as an HTML text node.

Figure 3: Structure of selectors and document nodes

4.1 Selectors, Documents and Edits

A computational document is a tree, consisting of four kinds of nodes (Fig. 3). 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 the same list element).

A reference is represented as a sequence of selectors (Fig. 3). 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 record. Because 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 behaviors are listed in Fig. 4. 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 can transform the document structure. Denicek tracks the effect of the edits on the structure and updates references when the document structure changes. Fig. 4 distinguishes between edits that keep references in a document unchanged (above) and edits that can 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

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

Figure 4: Summary of document edit types in Denicek

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 being dynamically typed). As a consequence, edits that transform the structure can serve both as structural edits (affecting the structure) and as value edits (affecting the value of specific nodes). The latter can be used, for example, when constructing an additional list item. An item is first constructed and then populated with values. This temporarily violates the invariant that collections are homogeneous (an issue that could be 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), an operation that should not affect references to the formula itself. To support value edits, edits that normally transform references (Fig. 4 below) have a *reference behavior* option that can 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. 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 4, below), Denicek updates the relevant references according to the rules shown in Fig. 5, provided that the reference behavior of the edit does not prevent updating.

Reference update behavior for WrapRecord and WrapList differs in that the updated references as a result of WrapList use the All selector. Although WrapList specifies the index to be used for the newly created list item, we assume that the operation introduces a homogeneous list, 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. (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 merge operation $\mathcal{M}_E(E_1, E_2)$ works on two edit histories, E, E_1 and E, E_2, that have a shared prefix E. Our merging 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. That is, $\mathcal{M}_E(E_1, E_2) = E, E_1, E_2'$. Note that the operation is not symmetric: $\mathcal{M}_E(E_2, E_1) = E, E_2, E_1'$. 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 edit history reconciliation takes two individual edits that occurred independently, e_1 and e_2, and produces a sequence of edits $e_2', e_2'', \ldots$ that can be applied after e_1 and have the effect of e_2, modified to respect the effects of e_1. There are two aspects of such reconciliation:

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

(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. 5, that apply when transforming references inside a document, although we now also support the case when $e1$ is Copy (see §B.2 for details).

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 5: 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 6: Effects of individual edit operations

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

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

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

If we want to append the edit e_2 after edits e_1, e_1', we need to update its target to reflect the 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 edits e_2', e_2'' that target the newly wrapped list and add the checkbox to the added item:

e_1 = **WrapRecord**(/todo, <div>, items)
e_1' = **Add**(/todo/items/*, done, **nil**, ☐)
e_2' = **Append**(/todo/items, #1, #0, Do some work)
e_2'' = **Add**(/todo/items/#1, done, **nil**, ☐)

If we performed the merge operation in the other order, we would 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 $\mathcal{M}_E(E_1, E_2)$ and $\mathcal{M}_E(E_2, E_1)$ result 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 is not symmetric. Such conflicting edits can be detected and reported to the user. Conflict detection is also used to implement incremental recomputation (§5.5), in which case conflicting edits produced by evaluation are removed and formulas have to be recomputed.

The Denicek substrate implements a conflict detection mechanism inspired by effect systems [68]. 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. Effects describe how an edit affects the document structure and we distinguish between three types of effects as shown in Fig. 6.
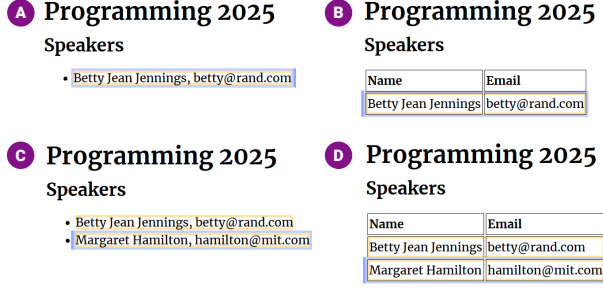
Figure 7: Merging of two independently done sequences of edits. Two ways of merging B and C result in the same D.
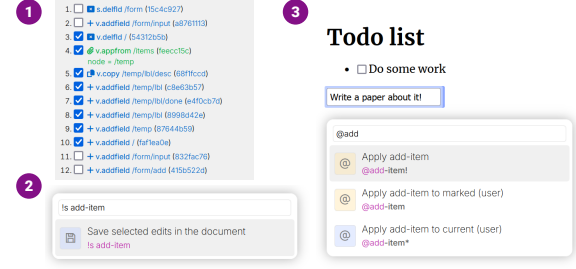

Figure 8: Programming by demonstration is implemented by selecting edits from the document history (1), saving them in the document (2) and replaying them (3).

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 $e'_2, e''_2, \ldots$ that can be added after E, E_1 and do not conflict with edits in E_1.

To do this, we iterate over edits e_2 from E_2 and check if the dependencies of e_2 conflict with effects of (1) any of the effect e_1 from E_1 or (2) effects of any of the previously removed edits. Here, the dependencies of e_2 include its target, but also 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 depend on the removed edit.

5 Programming Experiences

The key claim of this paper is that the Denicek computational substrate makes it easy to support a range of compelling user experiences. The experiences can be implemented by composing the primitive operations of the substrate, relying in particular on the operation for merging edits. We demonstrate this claim with the web-based Webnicek system, which uses Denicek to support:

(i) collaborative document editing (§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 experiences in isolation in the context of Webnicek in this section. The data science notebook system Datnicek, discussed in §7, provides a more comprehensive case study by combining multiple user experiences together.

5.1 Collaborative Editing

Denicek enables collaborative editing 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 reconcile edit histories. Merging requires coordination among users, but not a central server as in local-first software [53].

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 [27], 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).

Consider the scenario in Fig. 7. Here, the 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 the current implementation of our effect analysis, the two example histories 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.

5.2 Programming By Demonstration

In programming by demonstration [15], 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. 8), the user selects edits from the edit history, names them and replays them. For general-purpose document editing in Webnicek, this requires certain forethought, but as shown in §7, the mechanism is very effective in a domain such as data wrangling [45].

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 Denicek is needed and also that the stored edits can be modified by the user or tools working with the document.

Second, when replaying edits, Webnicek does not append the recorded edits on top of the current history. Instead, it stores the
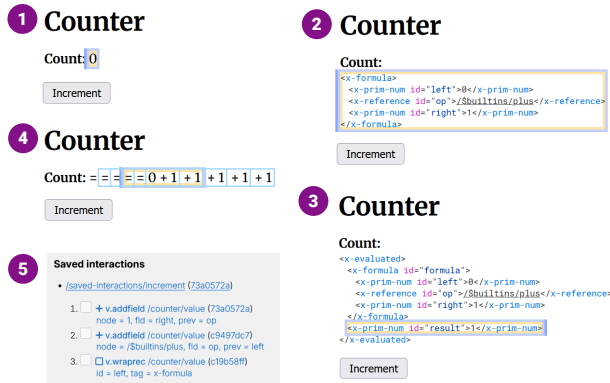
Figure 9: We wrap the initial count (1) in a formula that adds 1 to the value, (2). Evaluation produces the count (3), which is invalidated on subsequent clicks (4). The button replays the saved edits that wrap the current count in a formula (5).

hash of the history at the time of saving. To replay the edits, it then appends the edits to the top of the original history and merges this edit sequence with the current history. This pushes the recorded edits through later edits made by the user, reconciling them with potential structural edits. The result can be seen in Fig. 2 (E), where a newly added list item becomes 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. 10 (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 the previous section, 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 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. As shown in Fig. 10 (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,
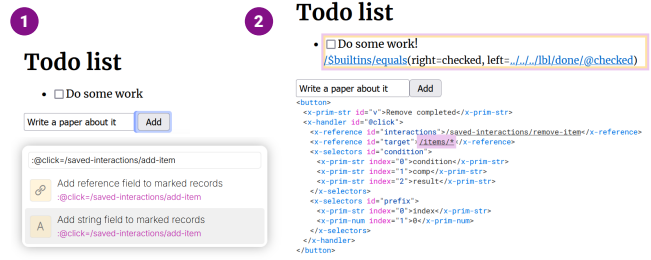

Figure 10: Using programming by demonstration to define a UI. The "Add" button (1) replays edits; the "Remove completed" button (2) modifies target and specifies a condition.

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 that edits cannot depend on values (§4.1). 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 heuristics for generalization [75] that infers and suggests such generalizations. If integrated into Webnicek, such heuristic could for example automatically construct a formula based on positive and negative examples [60] (selected and deselected Todo list items).

5.4 Formula Language and Evaluation

As illustrated earlier (§3, F), Denicek documents can contain formulas inspired by the spreadsheet paradigm [76]. 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 also 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 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 (5) are used as an event handler for the button and so clicking the button creates an increasingly long sequence of increments. One advantage of
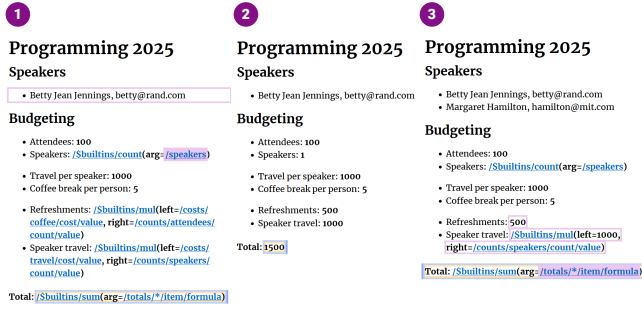
Figure 11: Budget calculation based on the number of speakers (1) and the result (2). When speaker is added (3), only the results of affected formulas are invalidated.
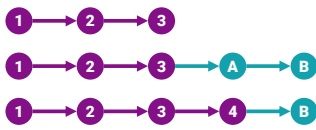

Figure 12: Evaluated edits (A, B) are kept at the top of the history. Evaluated edits (A) that conflict with ordinary edits (4) are dropped.

this approach that we will leverage later is that the evaluation also records a trace of how the final computed value was obtained.

> **x^2 Formula Language.** Webnicek exposes the underlying representation of formulas to the user, but the same representation could be edited through a user-friendly mechanism such as a textual calculation view [98], or a block-based editor [41]. The key point is that Denicek's tree structure makes it easy to embed formulas in document in a uniform way, edit them as other nodes and merge edits that change them.

5.5 Incremental Recomputation

As illustrated in Fig. 11, Webnicek 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. 12. When the formulas are evaluated, Denicek generates edits that wrap the formula in `<x-evaluated>` and set the result to the computed value. In Webnicek, 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 using edit reconciliation. If the edits conflict (according to the conflict detection discussed in §4.2), the affected evaluated edits are dropped.

The dependencies between edits alongside with the conflict detection mechanism provide functionality that other systems implement using an explicit dependency graph or by topological sorting.

> **↻ Live Programming.** The Webnicek prototype does not automatically evaluate formulas. This makes it easier to understand the evaluation mechanism, but a realistic system based on the substrate could automatically evaluate formulas to provide a live programming experience [85, 96] and use incremental recomputation for performance reasons.
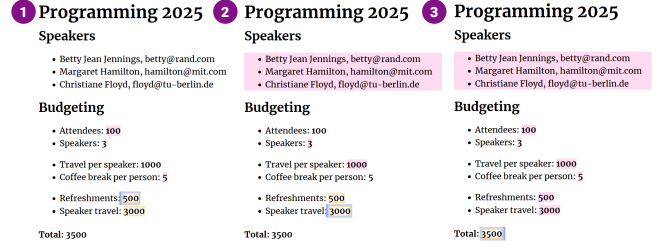

Figure 13: Using provenance tracking to highlight document parts that contributed to the calculation of refreshments costs (1), travel costs (2) and all costs (3).

5.6 Schema Change Control

Document structure often needs to evolve [11], 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 [93] and has recently been explored in the context of programming systems [27].

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 [27]. 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. 14. 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`.

Note that Denicek can only reflect schema changes explicitly represented by the document structure. If a formula depended on the structure of the original string representing speakers (with a comma), Denicek would not be able to amend the logic of the formula. Systems supporting the divergence model [27] may be able to lift this restriction by transforming the new value back before applying the original formula.

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).

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 [52]. One way to help users answer the question is to provide an explanation how a value was obtained [57]. Webnicek provides a basic mechanism that highlights document nodes that contributed to a specific computed result, illustrated in Figure 13.
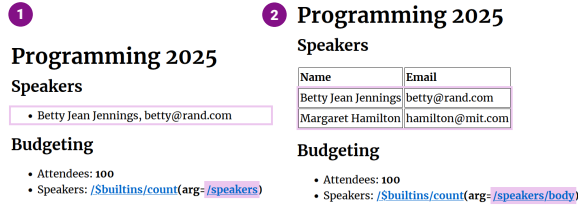
Figure 14: When an edit changes the document structure, references in formulas are updated accordingly.

The implementation leverages the fact that evaluation wraps the previous state of the formula (§5.4). When a formula is fully evaluated, the `<x-evaluated>` document node contains the result, but also a sub-tree with the full evaluation trace [82]. 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 can 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 [82] or to automatically construct a linked visualization [83].

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 is making programming more concrete [22, 103], i.e., to support copying of functionality as in prototype-based object-oriented programming [104, 111].

Webnicek supports a functionality akin to managed copy & paste [24, 26] 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.

The functionality is illustrated in Fig. 15, which uses the builtin operation `read-csv` to load a table from a file. The operation imports the data as a document table with rows that can be addressed by subsequent formulas. The then user writes two formulas to sum the rows and compute an average.

They copy a formula with switched operands using the Copy edit and then modify 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 operands of the division. When they merge the temporary fork into the original document, the *Apply to Newly Added* logic of the merge operation (§4.2) duplicates the RenameField edits and applies them to the copied formula. Merging with the Copy edit and the fact that formulas are ordinary document nodes, provide the key component for a straightforward implementation of the managed copy & paste functionality.
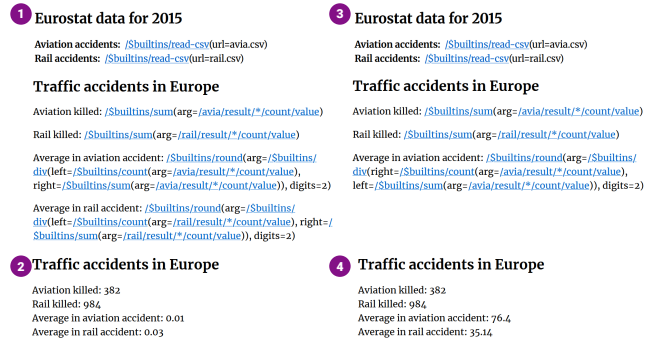

Figure 15: The user uses creates a copy of a formula (1). They notice an error (2), go back in history to switch the arguments (3), merge the change and re-evaluate both formulas (4).

> **📄 Linked Editing.** Webnicek currently requires users to explicitly manipulate history to correct errors across multiple code clones. Research on managing duplicated code resulted in multiple tools [21, 110] 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 detailed in §A. In this section, we document the design challenges, many of which are shared with related systems [23, 33, 40, 79].

> **🎲 Uniformity and Composability.** Two guiding principles for the design of Denicek have been *composability* and *uniformity* [39]. 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 change, a structure change, the result of user interaction or the result of formula evaluation.

Dynamic and Static Typing. Denicek does not explicitly track the document structure (it is dynamically typed). For example, we assume that lists are homogeneous, but do not enforce this property. A statically typed system could enforce such properties and explicitly 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.

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

Expressive Selectors and Edits. Our merge operation works on a pair of edit sequences, 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.

This limitation means that edits generated by generalized interactions (§5.3) can be only merged in limited ways (an item added to a list through merging after invoking an action will not be affected by the action). We expect that the restriction on expressiveness of selectors can be lifted for edits that do not modify the document structure, but have not yet lifted this restriction in Denicek.

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 (e.g., when adding a speaker or a Todo item).

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

To maintain order of list elements, Denicek uses a data structure inspired by the list MRDT [44] 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.

Structure and Capabilities of Merging. 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 [27] where users have to merge all changes in order and cannot adopt selected edits (cherry picking in git). The *divergence* model would let users keep their own schema but import all data edits. Supporting the model requires a *retract* operation [25] that is dual to our edit reconciliation (given subsequent edits e_1, e_2, 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). Add edits need to record additional dependencies identifying the source of the operation arguments. Alternative evaluation models avoid this need, but lead to a less uniform system design.

7 Case study: Datnicek Notebooks

Denicek is a low-level computational substrate. It is intended 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 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 [84]. This allows Datnicek to keep track of the code structure and potentially makes the system accessible to non-programmers [70]. We do not initially aim to implement advanced keyboard-based editing [5, 73].

- *Collaborative Editing.* It should be possible to merge independently done code changes as in Grove [3], addressing the known pitfall of versioning Jupyter notebooks [102].

- *Code Completion.* During editing, code completion should offer available data transformations and operations as when using type providers [106] or iterative prompting in The Gamma [87].

- *Output Invalidation.* When code is edited, the previously evaluated results that depend on it, directly or transitively, should be invalidated as in Wrattler [85, 89], addressing another well-known limitation of Jupyter notebooks [58].

- *Interactive Data Cleaning.* It should be possible to edit tabular data in an interactive grid and use programming by demonstration for common cleaning tasks as in Wrangler or Vizier [45, 50].

- *Wrangling Code Synthesis.* Interactively constructed data transformations should be convertible into code that can be checked by the data scientist and further edited as in Wrex [20].

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 [28], 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 trigger the same type of event that appends edits to the list of Denicek edits. Remaining events update the transient user interface state. The support for collaborative editing is implemented through an ad-hoc mechanism.

User Interface. Datnicek notebooks consists of a series of cells. In code cells, the user can use the "+" button to add a new variable,
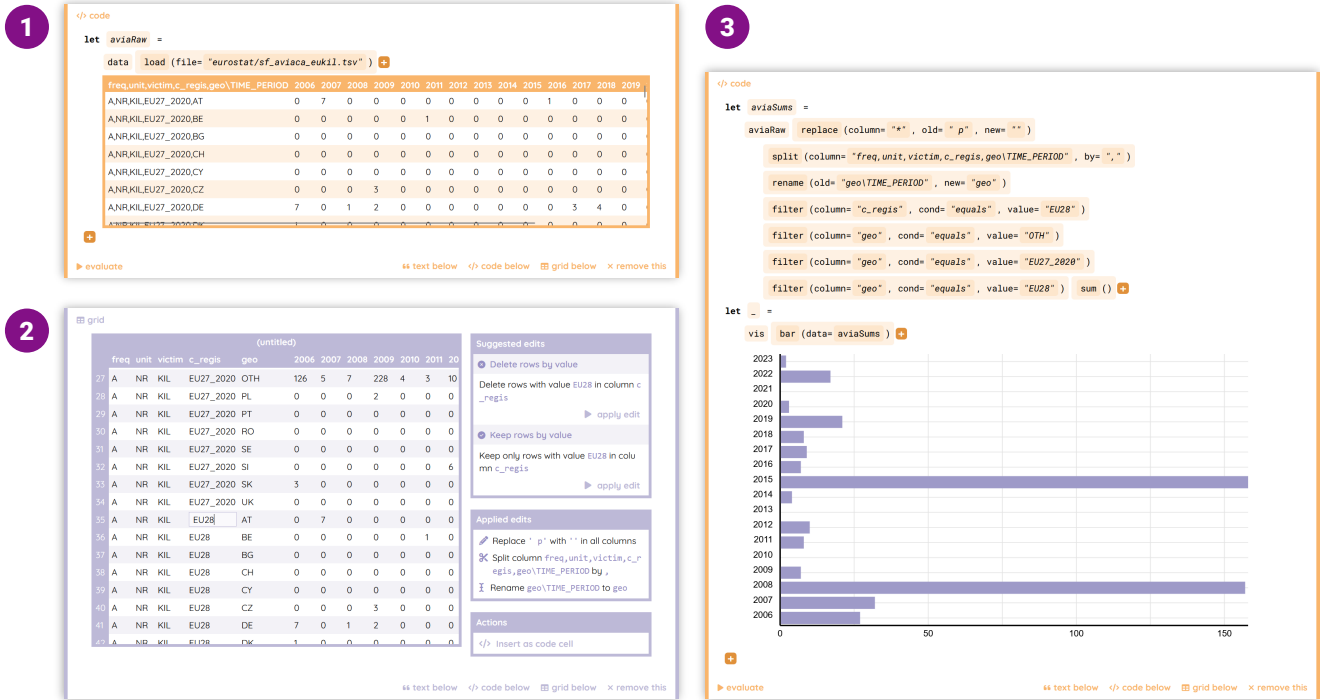
Figure 16: A notebook visualizing Air traffic accident data from Eurostat. The user loads data in a code cell (1) and edits it in a grid that infers edit operations via programming by demonstration (2). They turn the edits into a code cell (3) and add a chart.

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. 17. 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 [85]. A cell consists of a sequence of bindings of the form 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 $e.m(e_1, \ldots, e_n)$.

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 [86]. 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 single transformation may correspond to multiple underlying Denicek edits. For example, splitting a column named "foo/bar" with values of the form "7/5" 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 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 support conditional selectors. To implement 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

External Validity. The development of Datnicek was only started after the design and implementation of Denicek and Webnicek was completed. Thanks to this two-phase methodology, the case study provides a qualitative evaluation of Denicek's capabilities. The design of Denicek is based on third-party research (§7.1) and so the case study provides a limited external validation. However, several design choices of Datnicek align it with the capabilities of Denicek.

```
</> code

<cell-code>
  <x-formula id="aviaRaw">
    <x-prim-str id="file">eurostat/sf_aviaca_eukil.tsv</x-prim-str>
    <x-reference id="inst">/$datnicek/data</x-reference>
    <x-reference id="op">/$datnicek/data/load</x-reference>
  </x-formula>
</cell-code>

▶ evaluate          66 text below   </> code below   ⊞ grid below   × remove this
```

```
⊞ grid

<cell-grid>
  <edits id="edits">
    <transform index="0">
      <x-prim-str id="title">Rename columns</x-prim-str>
      <x-prim-str id="icon">fa-i-cursor</x-prim-str>
      <x-prim-str id="description">Rename column
      'freq,unit,victim,c_regis,geo\TIME_PERIOD' to
      'freq,unit,victim,c_regis,geo'</x-prim-str>
      <x-formula id="code">
        <x-reference id="op">/$datnicek/table/rename</x-reference>
        <x-hole id="inst">
        </x-hole>
        <x-prim-str id="old">freq,unit,victim,c_regis,geo\TIME_PERIOD</x-prim-str>
        <x-prim-str id="new">freq,unit,victim,c_regis,geo</x-prim-str>
      </x-formula>
      <edits id="edits">
        <x-edit-renamefld index="0">
          <x-selectors id="target">
            <x-prim-str index="0">*</x-prim-str>
          </x-selectors>
          <x-prim-str id="old">freq,unit,victim,c_regis,geo\TIME_PERIOD</x-prim-str>
          <x-prim-str id="new">freq,unit,victim,c_regis,geo</x-prim-str>
          <x-prim-str id="refs">update</x-prim-str>
        </x-edit-renamefld>
      </edits>
    </transform>
  </edits>
  <x-reference id="target">/cell-3e5d2ea3/aviaRaw</x-reference>
</cell-grid>

                    66 text below   </> code below   ⊞ grid below   × remove this
```

Figure 17: Underlying document representation of a code cell and a grid cell in the Datnicek notebook system.

This includes the use of a structure editor and a user interface for programming by demonstration that suggests edit operations. We revisit the generality of Denicek in §8.2, but establishing external validity ultimately requires adoption of the open-source Denicek package by other programming system researchers.

Effectiveness and Uniformity. Many of the Datnicek requirements are addressed by Denicek with minimal development effort. Denicek also provides a small and uniform set of primitives. In particular, the implementation represents all non-transient changes to a notebook as Denicek edits. Code completions offered by the code editor are edits (wrapping a formula or appending a parameter), changing a primitive value is an edit, and recommendations in the interactive grid are also edits (appending a `<transform>` node).

The uniform and dynamic nature makes the development experience of using Denicek more akin to dynamic object-oriented languages than to statically-typed functional languages. The system can quickly evolve, but it needs suitable debugging tools, such as the source view (Fig. 17) or an ability to step through the history.

Notebook Editing. Collaborative editing based on merging of edits makes conflicts less common when compared to textual merging. For example, changing a method parameter can be merged with adding a call to a chain. However, adding two independent calls still results in a conflict. A formula representation that avoids nesting [84] may eliminate an even larger proportion of conflicts. Variables are represented as references and automatic reference updating prevents a number of errors. References are automatically updated when the field (variable) name changes and Denicek prevents the deletion of cells that contain variables to which there are references.

The proof of concept nature of Datnicek means that it currently lacks suitable user interface for resolving conflicts. Similarly, the structure editor for code is pointer-based and would benefit from a better support for keyboard-based input [5, 73]. However, Denicek provides a suitable infrastructure for implementing this.

Evaluation Model. The evaluation model of Denicek implements dependency tracking and incremental recomputation, which aid notebook reproducibility [58, 89]. As in Webnicek (§5.5), Datnicek keeps evaluated edits on top of the current history and uses conflict detection to remove invalidated edits. We uses the same logic to evaluate edits performed in interactive grid cells, suggesting that the approach is a useful general implementation pattern.

However, as invalidation of evaluated edits is not built into Denicek, it is worth investigating if Denicek could support the standard evaluation model of Jupyter notebooks based on mutable state.

The development of Datnicek uncovered some limitations of conflict detection in Denicek. In particular, edits generated by evaluation (such as Add edits that set the result) need to be treated as conflicting with overlapping edits with both value and structure effects. This is because evaluated edits are the result of evaluation that considers both value and structure of source nodes.

Finally, data loaded in Datnicek is represented as Denicek nodes (a table is a list of records). Such uniform representation makes Denicek more versatile, but places a high demand on the system performance. We implemented a few basic Denicek optimisations, but Datnicek is still cumbersome to use with data consisting of multiple thousands of rows.

Future Work. One of the design choices discussed in §6 is whether to track the document structure explicitly. The structure of documents created in Webnicek can be irregular, which justified the dynamic nature of Denicek. In contrast, Datnicek notebooks are more regular. Replicating Datnicek using a system akin to Denicek, but based on explicit structure would yield an interesting comparison.

Our experience with conflict detection shows the importance of getting the details of the implementation of programming substrates like Denicek right. This could be aided by the development of a formally tractable model of the system and proving a correctness property akin to that of The Gamma live previews [85].

Finally, the representation of transformations in interactive grid cells (Fig. 17) highlights an unnecessary duplication – a transformation stores a sequence of edits, as well as a formula. Unifying these two notions is another interesting future direction.

8 Evaluation

The Datnicek case study provides a primary qualitative evaluation of Denicek. We complement this with heuristic evaluation according to the criteria proposed by Olsen [78] and through the technical dimensions of programming systems framework (TDPS) developed by Jakubovic et al. [39]. Criteria and dimension names are in bold.

8.1 Complex System Evaluation

First and foremost, Olsen [78] argues that a system must illustrate its **importance** for a particular class of users performing certain tasks in a given situation. Denicek is designed to support programmers and researchers developing novel interactive programming systems,

an active research area, arguably constrained by the dependence on existing programming languages and systems.

To support a wide range of novel systems, Denicek must satisfy Olsen's criterion of **generality**. We show that Denicek is suitable for building two very different kinds of programming systems, Webnicek and Datnicek. We further delineate the space of supported programming systems in the next section using TDPS.

Denicek makes it easy to try different solutions, satisfying the **flexibility** condition. This was experienced during the development of Datnicek, where a uniform representation (everything is an edit) made it possible to quickly prototype different designs.

For an emerging class of programming systems built around documents, such as Potluck [66] and notebooks for data science, Denicek provides **expressive match** by being nearer to the problem being solved, i.e., by using document representation for a programming system built around documents.

Denicek also satisfies the **inductive combination** criterion by offering a small set of primitives from which different designs can be built. Specifically, we used Denicek to build a document-based programming system, interactive grid for data cleaning and a structure editor with contextual code completion.

Several Olsen's criteria point to limitations of our work. First, it is unclear if Denicek can empower new participants to be involved in programming system design. Second, the Datnicek case study did not address scalability, although there is arguably a number of domains for which the capabilities of the system already suffice.

8.2 Technical Dimensions

The technical dimensions (TDPS) framework [39] maps the design space of interactive programming systems. We use the framework to characterize the generality of the substrate. For some dimensions, Denicek can cover the full range of options, while for others, it is limited to one fixed design or a subset of options.

Interaction. In both Datnicek and Webnicek, there is one primary **mode of interaction**. Both systems offer immediate feedback during editing, meaning there is one main **feedback loop**. Denicek could be used to build systems with multiple modes of interaction and feedback loops (e.g., by separating editing, checking of edits and evaluation), but it is unsuitable for live systems that preserve evaluation state during source code editing [10].

Denicek covers the full range of **abstraction construction** approaches. Abstraction *from concrete cases* can be implemented via programming by demonstration as shown in §5.2 and §7. Although Datnicek code cells do not currently support function declarations, these could be added to provide abstraction *from the first principles.*

Notation. Denicek supports both **notational structures**. *Complementing notations* use distinct representations for different aspects of a program. For example, Datnicek combines code cells with interactively constructed transformations in grid cells. *Overlapping notations* allow editing of the same underlying structure in multiple ways. Although not integrated in a single system, formula editor in Webnicek and code editor in Datnicek illustrate this option.

Even though Denicek's underlying conceptual structure is based on composability with a small number of primitives, this does not determine **uniformity of notations** of systems built on top of

Denicek. These can support representations formed by a small set of primitives, as well as by a large set of domain-specific constructs.

In Denicek, the **surface and internal notation** are typically closely related. The internal notation has an explicit structure (document nodes), which can directly encode the surface notation. Using an unstructured representation (such as text) is possible, but would be incompatible with other Denicek capabilities such as merging.

Conceptual Structure and Customizability. On the scale of **conceptual integrity vs. openness**, the Denicek substrate favors integrity. Systems based on Denicek need to use its representation of documents and document edits. This reduces complexity, but leads to incompatibility with existing software stacks.

Systems built using Denicek currently do not exhibit the **self-sustainability** property, i.e., the ability to be modified from within themselves. Supporting this is an appealing possibility. A further research on the Denicek computation model should attempt to make it possible to implement formula evaluation as, for example, conditional edits recorded in the document itself through programming by demonstration.

9 Conclusion

Many recently developed programming experiences make the task easier, less error-prone, more direct and more collaborative. We believe that many more research advances are possible. Alas, novel programming experiences are difficult to implement on top of existing programming languages. A suitable computational substrate can significantly reduce the implementation effort needed to implement novel programming experiences.

To this end, we present Denicek, a computational substrate that represents programs as sequences of document edits. We describe the design of Denicek and show how it supports a range of programming experiences. A remarkable property of the design is that many programming experiences are implemented through a straightforward combination of three basic operations provided by Denicek, namely edit application, merging of histories and conflict detection.

To evaluate the usability of Denicek for the development of interactive programming systems, we use it as the basis for Datnicek, a data science notebook system that supports a range of programming experiences pioneered in recent research systems. We also assess the substrate and its generality through heuristic evaluation.

The design of Denicek resolves a number of challenges that have been faced by authors of related systems, but have never been explicitly documented. While we hope researchers will adopt Denicek as a foundation for future innovative programming systems, an equally important contribution of our work is that it documents this existing but unwritten tacit knowledge.

Acknowledgments

References

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

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

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

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

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

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

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

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

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

[10] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! Continuous feedback in UI programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 95–104, New York, NY, USA, 2013. Association for Computing Machinery.

[11] 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.

[12] 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.

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

[14] Andrei Chiş, Tudor Gîrba, Juraj Kubelka, Oscar Nierstrasz, Stefan Reichhart, and Aliaksei Syrel. *Moldable Tools for Object-Oriented Development*, pages 77–101. Springer International Publishing, Cham, 2017.

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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. ACM, 2020.

[21] 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.

[22] Jonathan Edwards. Example centric programming. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '04, page 124, New York, NY, USA, 2004. Association for Computing Machinery.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

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

[47] 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.

[48] Stephen Kell. Unix, Plan 9 and the lurking Smalltalk. In *Reflections on Programming Systems: Historical and Philosophical Aspects*, pages 189–213. 2018.

[49] 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. ACM, 2024.

[50] Oliver Kennedy, Boris Glavic, Juliana Freire, and Mike Brachmann. The right tool for the job: Data-centric workflows in Vizier. *IEEE Computer Society Data Engineering Bulletin*, 45(3):129–144, 2022.

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

[52] 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. ACM, 2006.

[53] 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 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 154–178, 2019.

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

[55] Clemens Nylandsted Klokmose, James R Eagan, and P. 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

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

[62] 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. ACM, 2020.

[63] 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.

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

[65] 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.

[66] Geoffrey Litt, Max Schoening, Paul Shen, and Paul Sonnentag. Potluck: Dynamic documents as personal software. https://www.inkandswitch.com/potluck, 2022. Accessed: 2025-03-25.

[67] 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.

[68] 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.

[69] 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. ACM, 2013.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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. ACM, 2006.

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

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

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

[78] Dan R. Olsen. Evaluating user interface systems research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, page 251–258, New York, NY, USA, 2007. ACM.

[79] 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.

[80] 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.

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

[82] 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.

[83] 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.

[84] Tomas Petricek. Histogram: You have to know the past to understand the present. Presented at Workshop on Live Programming (LIVE'19), 2019.

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

[86] Tomas Petricek. Composable data visualizations. *Journal of Functional Programming*, 31:e13, 2021.

[87] 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.

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

[89] 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.

[90] Kevin Pu, Rainey Fu, Rui Dong, Xinyu Wang, Yan Chen, and Tovi Grossman. Semanticon: 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.

[91] 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.

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

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

[94] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. Babylonian-style programming: Design and implementation of a general-purpose editor integrating live examples into source code. *The Art, Science, and Engineering of Programming*, 3(9):1–39, 2019.

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

[96] 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.

[97] 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.

[98] 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.

[99] 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.

[100] 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.

[101] 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.

[102] 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.

[103] 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.

[104] 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.

[105] 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.

[106] 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. ACM.

[107] 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.

[108] Philip Tchernavskij. *Designing and Programming Malleable Software*. PhD thesis, Université Paris Saclay (COmUE), 2019.

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

[110] 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.

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

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

[113] 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. ACM.

[114] 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.

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

Counter App [51] – 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 [81] – 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, Collaborative Editing, Incremental Recomputation*

Conference List [27] – 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 [27] – 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 [72] – 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 [26] – 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 18: Formative examples used in Denicek design

A Formative Examples

The design the Denicek substrate, we identified six formative examples shown in Fig. 18. The examples range from established industry benchmarks (Todo and Counter apps) to cases from literature [26] and problems posed as schema change challenges [27]. 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 [20, 32, 60]. Our Hello World example is only a minimalistic illustration of such use, loosely inspired by earlier work [72].

B Merging Edit Histories

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

B.1 Apply to Newly Added

Assume that edits e_1 and e_2 occurred independently. We want to modify e_2 so that it can be placed after e_1. If the edit e_2 added new

nodes to the document that the edit e_1 would affect, we generate an additional edit that apply the transformation of e_1 to the newly added nodes (and only to those).

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

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

B.2 Transform Matching References

As above, assume that edits e_1 and e_2 occurred independently. We want to modify e_2 so that it can be placed after e_1. If e_1 is any of the three edits listed in Fig. 5 (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 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).