A Formative Examples

The design the Denicek substrate, we identified six formative examples shown in Fig. 17. The examples range from established industry benchmarks (Todo and Counter apps) to cases from literature [25] and problems posed as schema change challenges [26]. 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 [19, 31, 59]. Our Hello World example is only a minimalistic illustration of such use, loosely inspired by earlier work [71].

Counter App [50] – Counter with increment and decrement buttons.
The current count is represented by a formula that is modified by the buttons. The user can inspect the evaluation trace to see how the count was modified.
Programming by Demonstration, Incremental Recomputation, End-User Debugging

Todo App [80] – Buttons to add an item and remove all completed.
Adding an item must correctly merge with independently added functionality to compute which items are completed and remove them based on a formula result.
Programming by Demonstration, Local-First Collaboration, Incremental Recomputation

Conference List [26] – 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 [26] – 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 [71] – 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 [25] – Compute statistics using two data sources.
Formula to compute statistics can be reused with a different data source; error correction is propagated automatically to the copied version of the formula.
Concrete Programming, Incremental Recomputation

Figure 17: Formative examples used in Denicek design

B Merging Edit Histories

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

B.1 Apply to Newly Added

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

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

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

B.2 Transform Matching References

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