Denicek: Computational Substrate for Concrete, Collaborative, Interactive Programming

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

Abstract

Research on interactive programming systems gave rise to a range of compelling programming experiences, including local-first collaborative editing, programming by demonstration, incremental recomputation, schema and code co-evolution, provenance tracking and managed copy and paste. Those experiences advance the state of the art of programming, but they are hard to implement on the basis of existing programming languages and systems.

We contribute Denicek, a computational substrate that reduces the complexity of implementing the aforementioned programming experiences. Denicek represents a program as a series of edits that construct and transform a document consisting of data and formulas. Denicek provides two primitive operations on series of edits, merging and conflict checking, that form the backbone of the implementation of a range of programming experiences.

We discuss the architecture of Denicek, document key design considerations and elaborate the implementation of the programming experiences listed above. To evaluate the proposed architecture, we use Denicek as the basis of a simple innovative data exploration environment. The case study shows that the Denicek computational substrate provides an appealign basis for the design of richer and more accessible interactive programming systems.

Keywords

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

ACM Reference Format:
Tomas Petricek. 2018. Denicek: Computational Substrate for Concrete, Collaborative, Interactive Programming. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

A computational substrate defines the basic structures using which programs are constructed, how the progam state is represented and how it is changed [8]. 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 [10], while using lists as the basic substrate enabled Lisp to become a "language laboratory" [27].

In principle, any programming experience can be developed on top of any computational substrate. However, a program representation and operations for working with programs provided by a suitable substrate can eliminate much of the complexity of implementing an interesting programming experience. For example, the reflective capabilities of Smalltalk make it easy to provide rich debugging tools [3] that would be difficult to provide for C/C++ [11].

Programming Experiences. In this paper, we ask what would be the ideal programming substrate for making programming easier, collaborative and more transparent? We seek a programming substrate that makes it easy to develop programming experiences such as:

- *Programming by demonstration* – Allow non-programmers to construct simple programs by performing examples of the expected behaviour. [2, 18].
- *Local-first collaboration* – Multiple users should be able to use and modify a single program, preferrably without requiring a central server. [12]
- *Provenance tracking* – The execution of the program should leave an understandable trace that lets the user understand why program resulted in a particular result.
- *Schema evolution [extra-ish]* – When the user evolves the structure of the program, data and code should co-evolve automatically to match the new structure.
- *Notational freedom [extra-ish?]* – Allow users to adapt the program using a notation that suits them and is appropriate for the programming task at hand. [Joel]
- *Concrete programming [extra?]* – It should be possible to reuse parts of program or program logic without constructing abstractions, for example by managed copy & paste.[5, 6]

Methodology. citizen programmers?
DESIGN PROCESS = formative examples + evaluation case study

substrate as defined by [8]
[2, 18] [1]

[12, 13] [14–16] [21, 26] [22]

TODO: Maybe do not talk about “programs” because this is more documents with formulas - not that fancy programs. Computational documents?

explain computational substrate

JUSTIFICATION - this is a technical paper but belongs here!

1.1 Programming Experiences

list

1.2 Substrate

how it works very roughly

The coolest trick - almost everything (PBD, collaboration, evaluation) is done through merge!

1.3 Contributions

one main thing - substrate - with other things

2 Background

all the references

3 Walkthrough

To providea an overview of the programming experiences supported by the Denicek substrate, we use a web-based implementation of Denicek with a structure editor that supports navigation in document and issuing of edit commands. The editor implements support for a range of programming experiences, discussed in §5.

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

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

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

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

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

E *Merging Edits.* The two document versions can be automatically merged. The refactoring is applied to all existing speakers. New speakers added using the “Add speaker!” button are also automatically turned into the new format.

Collaborative Editing and Interactivity. Denicek's merging reapplies edits from another branch on top of the current history (§5.1). Merging is asymmetic, but the order does not matter here. The same merging operation is used when handling user interaction (§5.3).

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

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

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

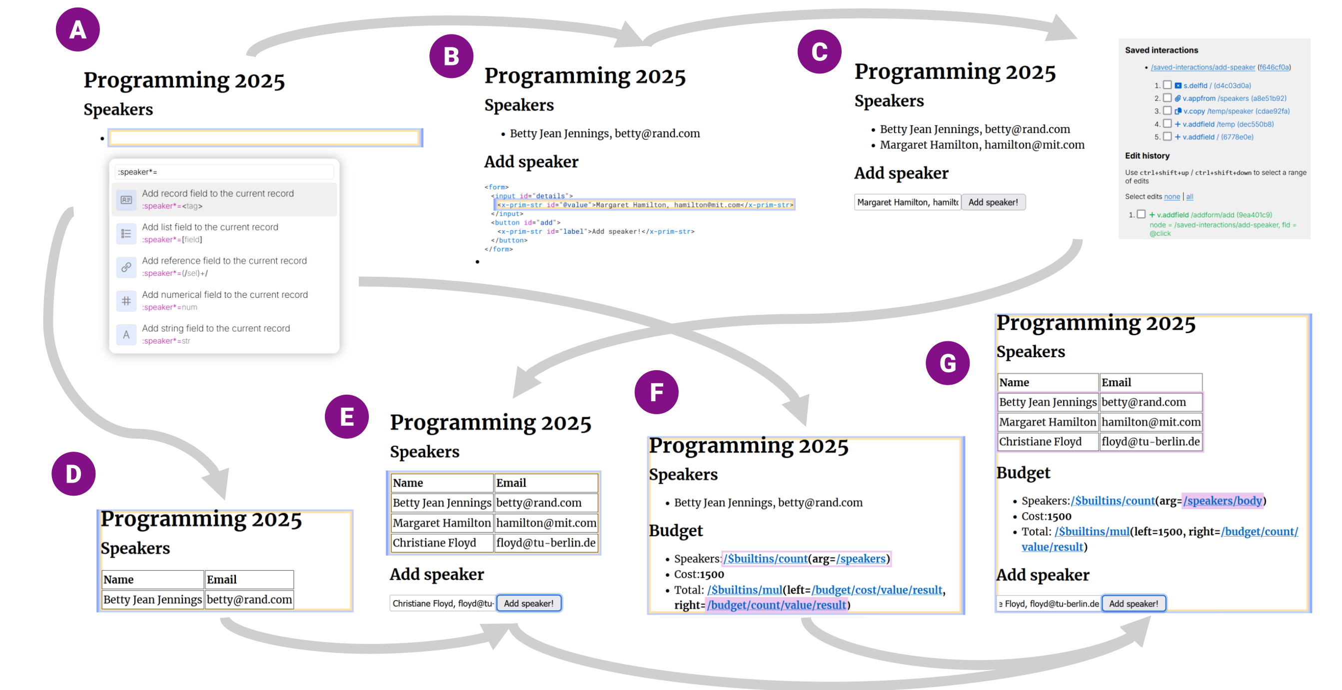

Figure 1: Organizing a conference using Denicek. The Walkthrough shows construction of a user interface for adding speakers (A, B, C); refactoring of the list and merging edits (D, E); and formulas with schema and code co-evolution (F, G).

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

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

Figure 2: Structure of selectors and document nodes

Schema Code Co-Evolution. References are understood by the substrate and so they are updated by structural edits (§5.6). The evaluation mechanism can replace formulas with values, but also augment them to keep an evaluation trace for 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 a wide range of programming experiences, discussed in §5.

4.1 Selectors, Documents and Edits

A computational document is a tree, consisting of four kinds of nodes (Fig. 2). Denicek follows the *naive realism* [4] principle and makes the entire document visible to the user, although parts can be collapsible or hidden using CSS. Records and lists are rendered as HTML elements of a specified tag with children becoming child elements. Field names are hidden in the rendered document.

References to document location are used in both document itself (reference nodes) and in edits (target of the edit). They are represented as a sequence of selectors (Fig. 2). The document model assumes that lists are homogeneous and records heterogeneous, and so the Any selector makes it possible to refer to all children of a list, but there is no way to refer to all children of a record.

Document Edits. The supported document edits and their behavior are listed in Fig. 3. All edits require *target* to which they are applied. Target is a reference and 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 particular kind.

The edits are designed to allow any transformation of a document through a series of steps whose effect can be tracked by the substrate. As illustrated earlier, Denicek updates references when document structure changes. Fig. 3 distinguishes between edits that keep existing references in a document unchanged (above) and edits that affect references (below). When a selector is *invalidated*, e.g., when deleting a field or a list item to which there is a reference, Denicek rejects the edit. Copying also invalidates selectors because it is ambiguous whether selectors referring to the original location should refer to the source or the target of the copying after the edit

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

Figure 3: Summary of document edit types in Denicek

(and references cannot refer to multiple structurally incompatible locations). Updating a field, wrapping, reordering or deleting a list item requires updating references in the document correspondingly.

Automatic Reference Update. There are two situations in which automatic update of references is undesirable. If an edit is applied to a singular element of a list (reference contains the Index selector), references that refer into any element of the list should be unchanged. Such edits may turn document into an inconsistent state, but they typically do so temporarily during document construction (checking such edits is discussed in §8.3).

Documents can also contain values that can be of multiple different kinds (i.e., a union type). In such case, references should not be updated when the kind of the value changes. For example, a formula may be either unevaluated or evaluated. As discussed in §5.4, evaluation involves wrapping, but this should not affect references to the formula. To support those cases, it is possible to annotate edits that normally affect selectors (Fig. 3, below) as non-structural. This annotation is required when the target reference contains the *Index* selector.

No Conditional Edits. An important aspect of the 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. (The effect of Copy depends only on the existing document structure, but not on its value; while AppendFrom affects only single list element and is thus labelled as a non-structural edit.)

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 has to 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 checked for conflicts or merged. Denicek identifies edit histories by a (git-like) hash, computed from the hash of the parent and the current edit, which is used to identify common shared part of the history during conflict checking and merging.

Applying Edits. When applying an edit, Denicek locates the target node and transform it according to the edit. If the edit may affect references in the document (Fig 3, below), Denicek updates matching references in the document according to the rules shown in Fig. 4 provided that (i) the edit is not explicitly marked as non-structural, and (ii) the target reference does not contain Index selector, i.e., it targets all elements of any involved lists. Also note that matching 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`. If the document contains a reference that would be invalidated by an edit, the edit is rejected.

Conflict Detection. Two edits are conflicting if the order in which they are applied matters. This is the case if they target the same node, or if one targets a node that is nested inside the node targetted by the other. For edits with dependencies (AppendFrom and Copy), a conflict also occurs if the other edit modifies the dependency.

Given two edit histories, one way to resolve conflicts is by removing (or marking as disabled) all edits from one of the histories that conflict with edits done by the other history. To do this, we first collect all targets of edits in the other history. We then iterate over edits from the first history to see if they depend on or target any of the affected targets. If an edit is removed, its target reference is added to the set of affected targets, so that other subsequent edits that depend on its original result are also removed.

Merging Edit Histories. Removing all conflicting edits is sometimes useful, but there are many cases where conflicting edits, in the above sense, can be reconcilliated. Say we have edit histories with the same common shared part E, E_1 and E, E_2. If the edits in E_1 and E_2 conflict, the results of applying E, E_1, E_2 and E, E_2, E_1 would be different. We can, however, construct edit histories E_1' and E_2' such that the results of applying E, E_1, E_2' and E, E_2, E_1' are the same.

The key operation that enables such reconcilliation takes two individual edits that occurred indpendently, e_1 and e_2, and produces e_1' that has the same logical effect as e_1, but can be applied after e_2. There are two aspects of such reconcilliation:

(1) *Apply Edit to Newly Added.* If e_1 is adding a new list item, but e_2 is changing elements of the target list, we apply the edit e_2 to the node added by e_1 so that, when it is added after the transformation, the added node has the new structure.
(2) *Transform Matching References.* If e_1 targets a node that is inside a node whose structure is changed by e_2, the target reference in e_1 is updated in a way that corresponds to the new structure.

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

DeleteItem *target, index* – Decrement Index greater than n in matching refs.
`/target/n/nested` ⇒ `/target/(n-1)/nested`

Reorder *target, permutation* – Update Index using permutation in matching refs.
`/target/n/nested` ⇒ `/target/permutation(n)/nested`

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

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

Figure 4: How document edits transform references

That is, using the same rules, shown in Fig. 4, that apply when transforming references inside a document.

Applying an edit to a newly added node involves a number of cases. When the new node is added using Add or Copy, but the target location is modified, this is an unresolvable conflict (because those two operations would overwrite existing nodes). When the new node is added as a new list item using Append, it can be transformed (by applying the other edit direclty to the new node).

Finally, when the new node is added using AppendFrom, it is copied from another document location. We cannot transform the node in the soource location (this would have unintended effects) or after adding it (we do not know its new index because of aforementioned *non-conditionality* of edits). Denicek reconcilliates such edits by first copying the source node to a new temporary location, transforming it there and then using AppendFrom from the temporary location.

5 Programming Experiences Implementation

The key claim of this paper is that the computational substrate described in the previous section makes it easy to support a range of experiences that make programming more concrete, collaborative and interactive. In this section, we describe how to use the substrate to support local-first collaborative editing (§5.1), programming by demonstration (§5.2 and §5.3), schema and code co-evolution (§5.6), incremental recomputation (§5.4 and §5.5), provenance tracking (§5.7) and concrete programming via managed copy & paste (§5.8). We describe the programming experience in isolation in this section. Next section provides a more comprehensive evaluation through a case study that combines multiple of them together.

5.1 Local-First Collaborative Editing

The Denicek representation enables *local-first* collaboration [12], as illustrated in §3E. If a document is edited by multiple users, they can each make edits to their local copy and eventually merge document variants using the operation to merge edit histories.

Merging of histories behaves akin to git rebase in that it keeps a linear history. Synchronization in a distributed system thus requires first reapplying local edits on top of the remote history, before updating the remote history. Denicek thus implements the *convergence model* of document variants [7], i.e., the user cannot, for example, maintain their own local document structure and import new data from another variant (this would require an inverse of the edit reconcilliation operation).

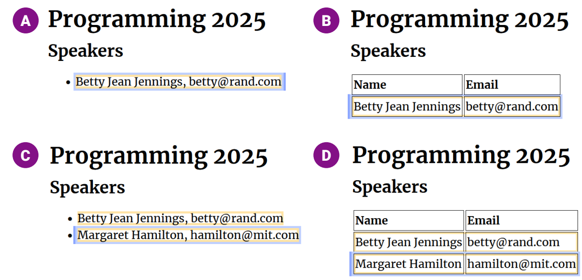

Figure 5: Merging of two independently done sequences of edits. Two ways of merging B and C result in the same D.

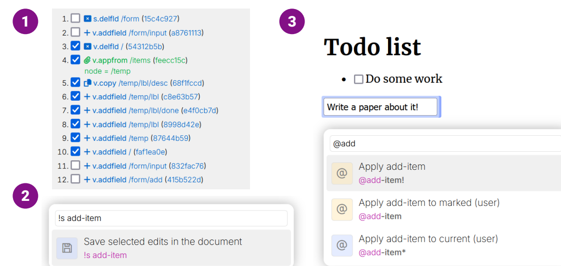

Figure 6: Programming by demonstration is implemented by selecting edits from the document history (1), saving them in the document (2) and replaying them (3).

Merging of edit histories is not symmetric. Document D in Fig. 5 can be obtained either by appending C' (produced by the edit reconcilliation operation) on top of A, B or by appending B' on top of A, C. Although the two edits are conflicting, C only affects data and B only affects structure and so the resulting document is the same in both cases. However, the histories differ. In the first case, the new node added by the Append edit is transformed (from primitive string to a record representing table row). In the second case, the structural transformations are automatically applied to all rows. As discussed in §4.2, conflicts during merging can be resolved either by removing conflicting edits or by letting the later edits overwrite the former ones.

5.2 Programming By Demonstration

In *programming by demonstration* [2], 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. 6), the user can select edits from the edit history, name them and replay them. In case of general-purpose document editing (in our formative prototype), this requires certain forethought (e.g., constructing a new list item in a temporary document field before appending it), but as illustrated in §7, the mechanism is effective in a restricted domain such as data wrangling [9].

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

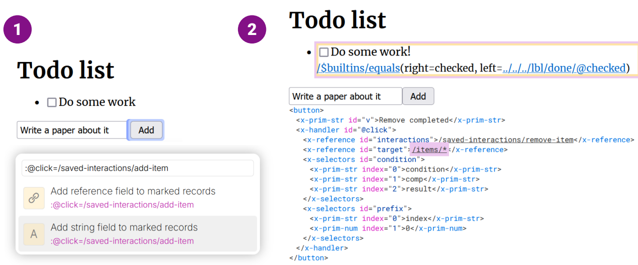

Figure 7: Using programming by demonstration to define a UI. The "Add" button (1) replays edits; the "Remove completed" button (2) modifies target and specifies a condition.

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

The implementation also has to account for the case where edits saved in the document are themselves transformed (when they are reconcilliated with other edits during merging). In this case, Denicek updates the saved edits (as discussed in §6, this would not be needed if the history was stored as a graph, akin to ordinary git merging rather than rebasing).

5.3 Interactive User Interfaces

Programming by demonstration can be used to define interactive elements in the document. In the simple case (Fig. 7 (1)), the `click` event handler is set to a reference to a sequence of edits saved in the document. Clicking the button executes the edits using the mechanism discussed in §5.2, i.e., by appending them to a history at the time of saving and merging them with the current history.

The Denicek substrate can be used to generalize the interactions saved through programming by demonstration. Our prototype illustrates this option in a limited way. As shown in Fig. 7 (2), a button to remove all completed TODO items can be generalized from the `remove-item` interaction, which removes the list item at the index 0. In addition to the saved interaction, we manually specify (in the source view) that the edits should be applied to all elements selected by the `/items/*` selector, instead of the original `/items/0` selector (prefix) and that the edits should only be applied to elements for which the formula (which tests if the checkbox is checked) specified by a relative selector `./condition/comp/result` evaluates to true.

Generalization Heuristic. Specifying generalization manually is cumbersome. Programming by demonstration systems typically implement heuristic for generalization [19] infers and suggests such generalizations. When integrated into a system based on Denicek, such heuristic may also be able to automatically add a formula based on positive and negative examples [17] (selected and deselected items).

5.4 Formula Language and Evaluation

Denicek documents can contain formulas inspired by the spreadsheet paradigm [20]. Formulas can specify richer computations

Figure 8: Increment wraps the existing count in a formula that adds 1 to the previous value (1), (2). Evaluation produces the count (3), which is invalidated on subsequent clicks (4).

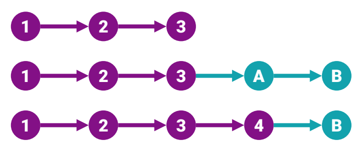

Figure 9: Evaluated edits (A, B) are kept at the top of the history. When they conflict with ordinary edits (4), evaluated edits are removed (A).

than what can be expressed using document edits. Formulas do not transform the document and their results are transient, although their evaluation also leverages the substrate operations, namely merging of edit histories.

As illustrated in Fig. 8, formulas are represented as document nodes with a special tag (`<x-formula>`). They are recognized by a formula evaluator and rendered in a special way (❶ and ❹), but they are created using ordinary edits and the Denicek substrate treates them as stnadard nodes.

To evalaute formulas, the formula evaluator generates edits that turn the `<x-formula>` record into `<x-evaluated>` (❷ and ❸), which keeps the previous formula state in the `formula` field and the evaluation result in the `result` field. (Keeping the previous state of the formula is not necessary, but it enables provenance analysis as discussed in §5.7.) The way edits generated by evaluation are merged with the document is discussed in §5.5

The counter example shown in Fig. 8 illustrates the interaction between formulas and programming by demonstration. To implement a counter, the Increment and Decrement buttons wrap the current counter value in a formula that adds or subtracts 1.

5.5 Incremental Recomputation

As illustrated in Fig. 10, Denicek supports incremental re-computation. In the example, the cost of speaker travel depends on the number of speakers, but the cost of refreshments depends only on two constants. Edit that adds a speaker only invalidates the former.

The evaluation mechanism is illustrated in Fig. 9. When the formulas are evaluated, Denicek generates *evaluated edits* that are appended to the top of the history. When subsequent edits are made, they are appended after all non-evaluated edits and the evaluated edits are pushed through the newly added edits. If the edits conflict (according to Denicek's conflict detection), affected evaluated edits and edits that depend on them are dropped.

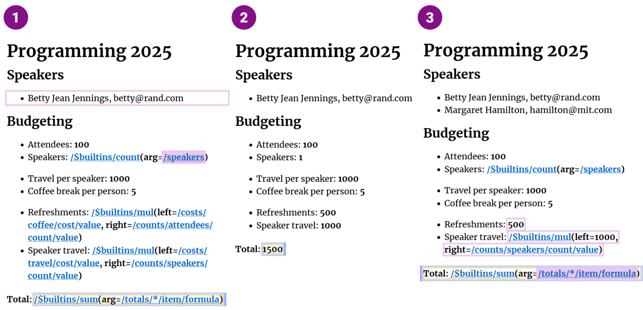

Figure 10: Budget calculation based on the number of speakers (1) and the result (2). When speaker is added (3), only the results of affected formulas are invalidated.

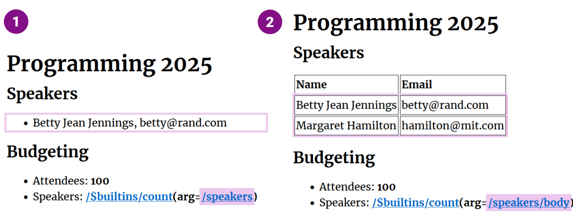

Figure 11: When an edit changes the document structure, references in formulas are updated accordingly.

> **Live Programming.** In the exploratory prototype, Denicek does not automatically evaluate formulas. This makes it easier to see and understand the evaluation mechanism, but a realistic system based on the substrate would likely automatically evaluate formulas to provide a live programming experience [23, 25].

5.6 Schema Code Co-evolution

The problem of schema evolution is that, when the schema used in a database or a program evolves, data and code that depend on the schema need to evolve correspondingly. The problem is well-known in database systems [24] and has recently been studied in the context of programming systems [7].

Although Denicek does not explicitly track document structure (schema or type), all documents have an implicit structure and some of the edits transform the document structure. As discussed in §4.2, Denicek automatically updates reference nodes in the document in this case. This enables a form of schema code co-evolution [7]. The formulas embedded in Denicek documents use reference nodes to refer to both data sources (in the document) and the results of other computations. Consequently, if the document structure changes, the formulas are automatically updated.

Consider the example in Fig. 11. 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`.

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

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

5.7 Provenance tracking

5.8 Managed copy & paste

6 Design Discussion

PRINCIPLES * merging of edits should not require looking at current document (non-computationality) * two ways of pushing edits should be equivalent, i.e.

push e1 e2 e3 over b1 b2 b3 can be done via push e1 over b1 = e1'; over b2 = e1"; over b3 = e1"' then push e2, then push e3 or push e1, e2, e3 over b1, then over b2, then over b3 (implies that no "extra" edits can be appended later)

USE MRDTs for list order!

DIFFICULTIRES * if you are not able to refer to newly added items, you cannot reasonably do PBD => need non-numerical indices

how to transform appends? * apply to added? (does not work for copy) * copy to doc and edit? (good because edits are isolated from other things, but messy code!) (if we edit in doc, it works and we also do not need indices for ListAppend) * keep nested edits for append op? also does not work, because of the tricky case

Tricky case is: - add new list item - then set its fields (this breaks things because if we push "add" through structural edits, they will not affect the new values added in the second step)

TODO: Delete ListAppend

// cf shadow dom v reactu // something like google forms!

TAG selectors - sound nice, but they break (you cannot tell if two selectors overlap e.g. when there is index >< tag)

save graph of edits rather than list (would make replaying of saved edits easier because the histhash never changes)

Although Denicek does not explicitly track document structure (or schema, or type), all documents have an implicit structure.

c.f. notes

memory mapped graphics etc.

unifying lists and records?

AppendFrom - hard to avoid! because recorded edits cannot know length (but having -1 in index would break too)

7 Case study

(data science environment)

8 Discussion

8.1 Heuristic evaluation

8.2 Limitations

8.3 Future work

use type system to check temporarily invalid state

ohter formal things * non-conditionality of edits * show that they can transform document from any to any without removing/readding values

PROPERTIES - can change any document to any other - sure, via remove add - but also more semantically (if it contained all values, can we do it without removing and adding them?)

maybe

xx

xx

introduction
background
related work
design process / goals
case study
formative research
formative study
analysis
system
implementation
evaluation / heuristic evaluation
discussion and limitations

9 Introduction old

The computational substrate using which software is built determines the capabilities that the software can provide. An imperative substrate that views programs as instructions modifying bytes in memory makes it almost impossible to allow end-user inspection or reprogramming of running software.

substrate as defined by [8]

A computational substrate defines what software is built from. This may be objects as in Smalltalk, lists as in Lisp, or memory with data and code as in UNIX/C. The different substrates enable different kinds of programming experiences. For example, object-oriented programming has historically been linked to the development of graphical user interfaces (where objects can correspond to elements on the screen). It has also enabled the development of visual programming environments such as the Alternate Reality Kit, based on message sending between objects.

In principle, any computational substrate can be used to develop any programming experience, but the greater the impedance mismatch between the substrate and the desired experience, the more difficult it will be to provide the experience and combine it with the rest of the system and other programming experiences developed for the system. (One can implement support for programming-by-demonstration using C/C++, for example as part of a game scripting engine, but it will not work with the rest of the ordinary C/C++ ecosystem.)

Medium is the message

9.1 Substrate

The question asked in this paper is, what would be the ideal programming substrate for supporting a range of programming experiences that make programs more collaborative, transparent and allows for a gradual transition from non-programmer to a programmer. We want a programming substrate that makes it easy to develop programming experiences such as:

- *Programming by demonstration* – Allow non-programmers to construct simple programs by performing examples of the expected behaviour. [18].
- *Local-first collaboration* – Multiple users should be able to use and modify a single program, preferrably without requiring a central server. [12]
- *Provenance tracking* – The execution of the program should leave an understandable trace that lets the user understand why program resulted in a particular result.
- *Schema evolution [extra-ish]* – When the user evolves the structure of the program, data and code should co-evolve automatically to match the new structure.
- *Notational freedom [extra-ish?]* – Allow users to adapt the program using a notation that suits them and is appropriate for the programming task at hand. [Joel]
- *Concrete programming [extra?]* – It should be possible to reuse parts of program or program logic without constructing abstractions, for example by managed copy & paste.[5, 6]

substrate as defined by [8]

[2, 18] [1]

[12, 13] [14–16] [21, 26] [22]

Joel's definition of substrate in Onward! Bret Victor talk https://www.youtube.com/watch?v=ef2jpjTEB5U

In what ways is a substrate "natural"?

thinglab - create line by cloning, it sticks to mouse pointer, clicking sticks it to something else squeak - has all the browsers (method search...)

computational substrate how it differs from computational media? more low-level - media suggests that there it comes

10 The whatever system

10.1 Document + Edits

defines

- selectors
- nodes
- edits

10.2 Walkthrough

* todo list? (or counter, but that is a bit boring)

11 Themes

* programming by demonstration - binding interactions to gui elements (event handlers) * provenance tracking - Amy Ko's whyline, Probe Log by HPI, enables linked visualizations * merging of edit histories / collaborative editing - bonus - can share restricted link to allow users fill out forms (allow partial edits only / def by selector?) * scehma change - change data & code accordingly * everything is an edit - interaction with the GUI - evaluation? tbd * copy & paste abstraction (requires finishing new approach to formulas!) - edit before copy to propagate edit to other places (or edit after copy to make it specific to a case) - higher order copying from https://tomasp.net/academic/papers/copy-paste/paint22.pdf * augmenters - cf. bonnie nardi (calls them something else - Jonathan says) - add programming by demonstration data wrangling gui to table (trigger interactions) cf. lorgnette

12 Applications

* todo list / counter / maybe too simple * (if used in the walkthrough, maybe something else? board game as in varv - tic tac toe? or 7guis?) * conference organizer * data exploration (ala histogram) * linked charts

13 Extras

* metablocks? * self-sustainability * some non-browser implementation of this (as in Varv?)

explicit structure self-sustainability notational freedom

Maybe have 'enabled' for edits afterall? (we can merge with conflicts and disable some edits, but keep them in history for info)

NOTES type Edit = Kind : EditKind Dependencies : Selectors list – only needed for evaluated edits

VALUE vs STRUCTURE distinction * good in theory, nice for implementation * tricky to use! needs some assistance tools

TODO - things to work on * "represent" edits somewhere in document as "library of functions" and then call those from buttons (rather than embedding them directly) allow some kind of abstraction (as in Histogram) to make them reusable * figure out how to do evaluation better (based on the stored abstractions? but need to store provenance...)

SEMANTIC CONDITIONS https://www.youtube.com/watch?v=NBnc2ToS_j0 (has a section on this in background)

SUBSTRATE DESIGN PROBLEMS * selectors - all for structure / index for data (but it is useful to allow others...) (multiselect also bad for checks!) * groups/conditions/preconditions (c.f. email to jonathan) tried conditions on edits; trying groups with check edits * what to do with "disabled edits"? for example when we remove all checked (before, this created edit groups with "check" but if the check was false, the group was ignored and this messed up merging - because we wouldn't know if the edit had any effect or not)

Evaluation * evaluated edits have to be migrated to the end (if there are conflicts, they are dropped) Think of this as maintaining a tree:

e3 | e2 evaluated | / e1 | e0

this has to be serialized as e0 -> e1 -> e2 -> e3 -> evaluated

evaluated edits do not became part of the main history but hang on the side

ISSUES * if we merge a thing with saved-interactions with something, hashes will change!

NOTES * ListAppendFrom - we need this, because we cannot encode this. * for records, we can RecordAdd(sel, fld, ..) @ Copy(sel @ [Field fld], src) but this does not work for lists - because we do not know the index! (and we cannot look into current document, because it will differ for saved-interactions)

TODO * many things with <tag> selectors currently do not work (e.g. 'matches' for highlighting) because if we collect path of a current node, we collect indices and get /some/2/another - and cannot tell if this matches /some/<li>/another - we'd have to collect more detailed path info!

INTERACTION * replay stored event handlers against the old version? (this way, adding an item to a speakers list gets migrated & adds a new table row!) * simlarly!! we need merge in order to apply edits to multiple targets (when you remove all items in a list, the indices change) (but I guess we should do this against version at the time of saving too....)

[this & evaluation = the unreasonable effectiveness of merging]

Notes on storing and reusing edits * references need to be represented as references so that they get updated (NO! not if we reply them against old version, which seems better - but there are 2 design choices) * how to apply them to multiple targets? use Move to update the selectors instead of replacing the prefix manually

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

CONDITIONALS https://toby.li/files/p311-radensky.pdf

REMAINING IMPLEMENTATION TODOs:

* Some kind of provenance visualization * Some kind of matchers/transformers mechanism (ideally to add interactive buttons to tables) * Apply to all (remove completed in TODO)

append /speakers #hamilton <li> addfld /speakers/#hamilton speaker "Margaret Hamilton, hamilton@mit.edu"

»> wraprec /speakers body <table>

[update selectors - add /body] append /speakers/body #hamilton <li> [dtto] addfld /speakers/body#hamilton speaker "Margaret Hamilton, hamilton@mit.edu"

»> retag /spekers/body <tbody>

[na - out of scope]

»> retag /speakers/body/* <tr>

[apply to added - update node (* => #hamilton scoping)] append /speakers/body #hamilton <li> + retag /speakers/body/#hamilton <tr> [na - out of scope] addfld /speakers/body#hamilton speaker "Margaret Hamilton, hamilton@mit.edu"

»> wraprec /speakers/body/*/speaker value <td>

append /speakers/body #hamilton <li> + retag /speakers/body/#hamilton <tr> + wraprec /speakers/body/#hamilton/speaker value <td> [no op - it is not there] addfld /speakers/body/#hamilton speaker "Margaret Hamilton, hamilton@mit.edu" + wraprec /speakers/body/#hamilton/speaker value <td>

» updid /speakers/body/* speaker=>name

append /speakers/body #hamilton <li> + retag /speakers/body/#hamilton <tr> + wraprec /speakers/body/#hamilton/speaker value <td> [no op - it is not there] + updid /speakers/body/#hamilton speaker=>name [no op - it is not there] addfld /speakers/body #hamilton speaker "Margaret Hamilton, hamilton@mit.edu" + wraprec /speakers/body/#hamilton/speaker value <td> + updid /speakers/body/#hamilton speaker=>name

»> addfld /speakers/body/* email ""

append /speakers/body #hamilton <li> + retag /speakers/body/#hamilton <tr> + wraprec /speakers/body/#hamilton/speaker value <td> [no op - it is not there] + updid /speakers/body/#hamilton speaker=>name [no op - it is not there] + addfld /speakers/body/#hamilton email "" [no op - it is not there] addfld /speakers/body #hamilton speaker "Margaret Hamilton, hamilton@mit.edu" + wraprec /speakers/body/#hamilton/speaker value <td> + updid /speakers/body/#hamilton speaker=>name + addfld /speakers/body/#hamilton email ""

...

copy /speakers/body/*/name => /speakers/body/*/email edit /speakers/body/*/name (fn1) edit /speakers/body/*/email (fn2)

References

[1] 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.

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

[3] Stefan Ramson Jens Lincke David Rauch, Patrick Rein 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.

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

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

[6] 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.

[7] 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.

[8] 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*, pages 240–258, 2022.

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] 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*, pages 1569–1578, 2009.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] 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.