We thank reviewers for their comments.

We are encouraged that reviewers find the scope of the work exciting (R1), timely (R2) and inherently ambitious (R3). We particularly appreciate that reviewers recognize our work as a “solution to a hard engineering problem” (R1), presenting “technically solid and thoughtfully constructed system” (R2) and recognize a more basic “insight that it’s worth looking for such a unified set of mechanisms” (R3). The reviews identify several specific areas for improvement:

\* The chief difficulty of our work is developing a design where all aspects fit together (appreciated by R3 with “ooo-ing and ah-ing”). R1 asks about core takeaways – as recognized by R3, our contribution is a design of programming system foundations (computational substrate) that support a wide range of programming experiences through a unified set of mechanisms. We will clarify the takeaway in the introduction and provide a better overview of the “programming system foundations” research agenda in the background to make the paper self-contained (re R2) by expanding the discussion on systems with similar aims (Subtext, BootstrapLab and Infra).

\* Multiple reviewers identified the need for clearer framing. We will include an overview figure on page 1 illustrating the system scope using two examples (TODO list in Webnicek and data analysis in Datnicek). The figure will show the relationship between the systems (re R3) with Denicek as the foundation behind the other two. It will elucidate our notion of a program and programming (re R2) and provide concrete opening examples (re R2).

\* R1 asks about distinction from prior work (especially Grove [3]) using tree representation. We will expand the discussion in the Background. Grove is a formalization of a structure editor, whereas Denicek provides foundations for a broader range of programming experiences (including structure editing and collaboration, but also PBD, re-computation, interaction and schema change control). Grove patches (construct, delete, relocate) are suited for its purpose but lack operations our formative examples rely on (copying, working with collections). We also rely on explicit (operation transform) edit reconciliation for PBD (Sec. 5.2) and re-computation (Sec. 5.6). Our abstractions are not novel as such. What is novel is that our single set of abstractions supports a combination of a variety of user experiences. Designing similar abstractions based on CRDTs is an interesting, but challenging problem. Formalizing our abstractions is also an appealing direction for our work (likely more suited for POPL than UIST).

\* Reviewers also ask about the concrete problem addressed (R2) and what makes end-user programming tools hard to build (R1). The latter question (aptly phrased by R1) deserves a deeper investigation. We believe the difficulty emerges when combining multiple programming experiences in a single system and from the need to move between the concrete and the abstract. This belief is based on our personal experience developing multiple systems (to be cited in the revision) and on a technical dimensions analysis (Sec. 8.2). A substrate based on edit history provides a middle ground between the abstract and the concrete and we show that it can be carefully designed to serve as the basis for a range of experiences.

\* R2 points out the lack of comparative baseline or user feedback and R1 wonders if others would find our design useful. As recognized by the UIST Author Guide, evaluation of complex systems is challenging. Denicek is intended as a foundation for system development. Learning and using it is a significant undertaking (possibly weeks, rather than days), which makes studies difficult to conduct. R1 wonders if the substrate was useful for our scenarios because its abstractions were tailored to the particular interfaces we were developing. The formative prototype (Webnicek) indeed informed the design of our abstractions, but the case study (Datnicek) was conceived later, after the abstractions were developed (see 7.3). It uses, for example, a different approach to PBD (see 7.2). We expect that an extensive evaluation can only be obtained by considering subsequent uses of the system. We will release (and intend to maintain) the system as open-source.

\* We are grateful to R2 for their detailed analysis of what makes PBD systems hard to build. It is worth noting that our design is a careful result of tradeoffs (for example, a richer mechanism for specifying targets is at odds with support for schema change control). Those tradeoffs will be made explicit in the revised paper. However, Denicek provides a foundation upon which richer mechanisms can be built (illustrated by the, admittedly still basic, example in Sec. 5.3).

\* R3 asks multiple specific expert questions. We will address those in detail in the revision, but include brief responses here. Denicek supports a richer set of values and operations in the formula language (Sec 5.8 uses primitive operation to read a CSV file; this will be shown in the revised version). However, integrating edits and formulas remains an open problem (noted in Sec 7.3). Throwing out conflicting edits is only done automatically for incremental re-computation, i.e., for edits resulting from formula evaluation (Sec 5.5). Our prototype does not include a user interface for resolving conflicts, but we expect that the user should make the decision explicitly (which is allowed by our OT-based design). The concrete concern involving edits depending on values is correct – we believe the restriction can be lifted for edits that do not modify document structure and will point this out in the revision. Finally, schema change control works for explicitly represented schema but not for implicit schema. A structured string value is an excellent example that we will include. (Systems supporting the divergence model of [26] may be able to lift this restriction.)