Correct-by-Construction Interactive Software: From Declarative Specifications to Efficient Implementations

Kyle Headley Matthew A. Hammer
University of Colorado at Boulder

Introduction. Today's interactive software is wildly successful and pervasive, invading nearly every facet of our daily lives. An interactive system, like a GUI, is also particularly complex, due to the nature of providing a rich, responsive, interactive experience for users.

Today's software platforms further compound this complexity for humans and tools alike because they force the programmers to design their application behavior in terms of *implementation concepts*, rather than *declarative specifications*. Typically, framework models for applications are complex: they consist of global, higher-order state and control-flow that is rarely direct, but often contorted into callbacks or continuation-passing-style (e.g., to be event-driven, and/or asynchronous) [5]. Seemingly, this complexity is required for the framework to be responsive and flexible.

However, designing user interaction based solely on such application models is inherently flawed, since the resulting design process omits a key element: a declarative specification. A declarative specification is most useful to its designers and users alike when it is not framed within the arcane ontology of the latest software framework, but rather given as a simple executable program that is *independent from the underlying framework implementation*.

As an alternative to framework-driven design, this extended abstract advocates a design process that we call *Correct-by-Construction Interaction*. Many of the underlying principles behind this process are not new, going back to pioneers such as Dijkstra, who famously advocated that programs and specifications should be developed hand-in-hand, and who proposed starting with a specification, and systematically deriving an implementation that is correct by construction. Our contribution to this tradition consists of the following insight: A promising bridge between declarative specifications and efficient, responsive applications are *general-purpose programming language techniques for incremental computation*.

To explore this connection, we designed and implemented a prototype text editor, IC-Edit. After sketching a specification, we implemented it in about 400 lines of OCaml, without any special libraries[1]. We explored this working prototype to resolve non-obvious design choices (e.g., "In the presence of multiple cursors, when does editing affect the placement of inactive cursors?", "How does undo/redo affect past input modes?").

Below, we summarize the features of IC-Edit, providing a video demo for further details. Next, we tour the formal specification of IC-Edit, on which our OCaml prototype is based. The prototype is simple to understand, but very inefficient for large workloads. We explore these problems in further detail, highlighting the sources of this inefficiency. Finally, we point to a promising solution, based on recently-proposed incremental computation techniques.

Features of IC-Edit. In addition to common editor features like insertion and deletion, IC-Edit includes an undo buffer tracking the entire history of the document. We have also included the ability to add additional cursors to the document. The writer makes use of only one *active cursor* at a time, but can switch between cursors at will. This can emulate multiple users, or repositioning the cursor within the text. IC-Edit has four input modes, for permutations of: insertion vs overwriting, and left-to-right vs right-to-left cursor direction. For a tour of IC-Edit's features, see these videos:

https://github.com/cuplv/icedit-demo

Formal Specification of IC-Edit. To formalize the interactive behavior of IC-Edit, we first codify the concepts of user actions and editing commands. Editing commands modify the text buffer and manage the pool of available cursors. A user action a consists of an editing command c, or the actions that undo and re-do past commands (undo and redo):

User Action	a	::=	cmd c | undo | redo
Edit Command	c	::=	ins $d\ t$ | rm d | repl $d\ t$ | mv d
		|	mk α | sw α | jmp α | join α
Direction	d	::=	L | R
Text symbol	t	::=	$\cdots$
Cursor symbol	α	::=	$\cdots$

Formally, an editing command c consists of inserting, removing or replacing text symbols (ins, rm, repl) or moving the active cursor within the text (mv), making new cursors (mk), switching among existing cursors (sw), jumping

[1] We use OCaml's standard `Graphics` library to perform interactive I/O.

to existing cursors (jmp), or joining the active cursor with an inactive one (join).

A *symbol sequence* S consists of cursor and textual symbols, and a *symbol zipper* $\langle S_1 \parallel \alpha \parallel S_2 \rangle$ consists of symbol sequences to the left (S_1) and right (S_2) of the active cursor α:

$$\begin{array}{rrcl} \text{Symbol sequence} & S & ::= & \epsilon \mid S :: t \mid S :: \alpha \\ \text{Symbol zipper} & Z & ::= & \langle S_1 \parallel \alpha \parallel S_2 \rangle \end{array}$$

Using these structures, zipper-transforming semantics for single commands c and command sequences C are given by the judgement forms $Z_1 \vdash c \longrightarrow Z_2$ and $Z_1 \vdash C \Downarrow Z_2$, respectively. These semantics are directly inspired by the *zipper pattern*, which provides a general approach for describing persistent (applicative, purely functional) data structures that undergo small changes. First proposed for n-ary trees by Huet, the zipper pattern has since been adapted and generalized extensively [1, 4, 6].

We use a second instance of the zipper pattern in the semantics for running an action sequence A, defined by the judgement $A \Downarrow C_1 \; ; \; C_2$; we read this as "performing action sequence A yields command history C_1, and undo buffer C_2." Based on the outcome of these actions, $\langle \epsilon \parallel \alpha \parallel \epsilon \rangle \vdash \mathsf{rev}(C_1) \Downarrow Z$ defines the final state of the buffer Z after command history $\mathsf{rev}(C_1)$. (The zipper representation for C_1 uses reverse order, from most to least recent commands).

The full specification for IC-Edit requires five judgements and 25 inference rules in total; it is available online:

https://github.com/cuplv/icedit-calc

Performance Challenges. Figure 1 illustrates a major performance challenge for IC-Edit: As the number of past actions grow, the time required for IC-Edit to respond after each action grows super-linearly. To produce Figure 1, we randomly generate a list of actions as follows: half of the actions are insertion commands, 20% are remove commands, most of the remaining actions consist of replace and cursor movement commands. Only 1% of random actions are undo actions, or commands that create, switch and jump among cursors. Under this random distribution of actions, we observe that after 100k actions, the response time is 120ms, which gives a noticeable lag. At 200k the response times degrades to 0.7 seconds, and at 500k it is over 4 seconds.

Switch and jump commands, though infrequent in this workload, are the most expensive for IC-Edit to process since it does so naively. Specifically, each such action requires linear time. Further, since IC-Edit naively reprocesses the entire history after each action, this results in a quadratic trend, as the plot shows. Without these cursor commands, the plot is linear, taking only 0.12 seconds per action with a history of one million previous commands.

The quadratic and linear times mentioned above are both undesirable. Ideally, the plot would be *sublinear*, even in the presence of cursor management (switching and jumping).

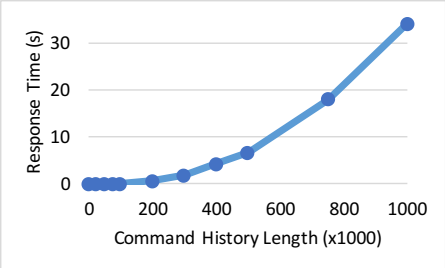

Figure 1. Response time versus command history length.

Specifically, we want an efficient implementation of IC-Edit to process each user action in $O(\log |A|)$ time, where sequence A is the action history.

Proposed Solution: Incremental Computation. To bridge the performance gap between the simple prototype in OCaml and an efficient implementation, we propose using Adapton, a language-based technique for incremental computation [2, 3]. Under the hood, Adapton employs memoization and dependency graphs to record past computations and selectively reuse their results. We plan to use Adapton's notion of memoization to improve the efficiency of repeatedly reprocessing the command history, and searching for cursors within the editor's content. With this approach, we aim to achieve the $O(\log |A|)$ response time mentioned above.

However, applying memoization is not automatic. It requires semantics-preserving transformations that alter data structures (changing linked lists into trees) and the structure of recursive operations (changing tail recursion into recursion over balanced trees) [2]. We are in the process of applying these transformations now; we expect preliminary results in the coming weeks.

References

[1] Michael Abbott, Thorsten Altenkirch, Conor McBride, and Neil Ghani. D for data: Differentiating data structures. *Fundam. Inf.*, 65(1-2):1–28, 2004.

[2] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. Incremental computation with names. In *OOPSLA*, 2015.

[3] Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. In *PLDI*, 2014.

[4] Gérard Huet. The zipper. *Journal of Functional Programming*, 1997.

[5] Neelakantan R. Krishnaswami. Higher-order reactive programming without spacetime leaks. In *International Conference on Functional Programming (ICFP)*, September 2013.

[6] Norman Ramsey and João Dias. An applicative control-flow graph based on Huet's zipper. *Electron. Notes Theor. Comput. Sci.*, 148(2):105–126, 2006.