Please Repeat Yourself

A Simple and Responsive Text Editor via Incremental Computation and the Random-Access Zipper

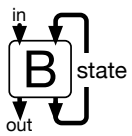
Blinded

Abstract

Today's interactive software is complex to write and read. Arguably, much of this complexity stems from callback-based programming models, which in turn rely on global state with complex, global invariants. In this paper, we propose an alternative approach, the "Please Repeat Yourself" (PRY) methodology for designing and implementing interactive software. PRY first encourages programmers to decompose interactive behavior into pure functions that, at each instant in time, compute the system's current output by processing the system's *entire input history*. After this simple (but naïve) design, the methodology refines the prototype with techniques for general-purpose incremental computation (IC), resulting in a responsive implementation.

As a proof of concept of PRY, we present IC-Edit, a primitive text editor with basic end-user editing features. The features of IC-Edit consist of cursor-local edits, non-local navigation, and a global undo/redo buffer. In the context of IC-Edit, we demonstrate that the PRY methodology leads to both a simple model, as well as a responsive implementation. In particular, our experimental evaluation shows that IC-Edit responds on average in under 8ms. In general, its response time is sublinear in the total number of user actions, whereas a naïve implementation of its model has quadratic response time. IC-Edit achieves this asymptotic improvement by employing novel data structures and algorithms for incrementally-edited sequences; these contributions generalize list zippers with efficient random access.

1. Introduction



Software that interacts with users is complex to specify and implement because these systems store *state* that persists and evolves in time. The left figure illustrates the prototypical behavior B of an interactive system at the highest level of abstraction. At successive instances in time, the system receives new input from the user or environment (in), produces new output (out), and feeds its state back to itself for future interactions (state). The complexity of interactive systems arises as they strive to implement *responsive* behavior that quickly computes the latest output from the latest input and state. In particular, responsiveness comes at the price of complex representations of, and invariants about, the system's state.

In this paper, we propose a radical alternative to the current design methodologies for building interactive systems; we refer to this as the "Please Repeat Yourself" (PRY) methodology. The first phase of PRY seeks a *simple* system model by making two non-traditional design choices, depicted below. The first design choice decomposes an interactive system's behavior, separating its state updates from a purely-functional computation maps the current state to the output (viz., `update` vs `compute` below). The second design choice selects the *simplest possible* definition of system state: *the history of all user inputs*. Consequently, updating the system's state merely consists of using `Cons`, which prepends the current input to the list of prior inputs:

1. Decompose `behavior`

```
behavior(in, st) =
  st' ← update(in, st);
  (compute(st'), st')
```

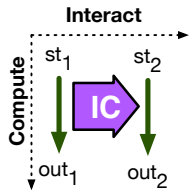
2. Define `update` := `Cons`

```
behavior(in, st) =
  st' ← Cons(in, st);
  (compute(st'), st')
```

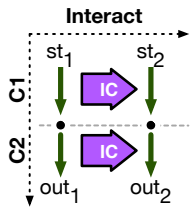
This definition of system state is justified by the desire for simplicity, and the desire to interact with users, where systems commonly offer "undo" and "redo" actions. In these contexts, maintaining a history of past inputs, perhaps up to some limited threshold, is a necessity. Though some systems impose a threshold on the length of this history for practical reasons, we assume throughout this paper that the goal is to offer *unlimited* user-controlled backtracking; future work should explore how more nuanced trade-offs affect the proposed approach. In sum, PRY encourages system designs where the complexity of the system state is as low as possible (viz., the history of all past inputs), and where the burden for responsive interaction rests on a clever, *purely-functional* implementation of `compute`, which transforms the full input history to the current output.

Based on the simple model from the first phase, the second phase of PRY seeks a *responsive* system model. We observe that, by definition, `compute` performs redundant work for each state update. Consequently, by expressing `compute` as a purely-functional algorithm, we can system-

atically exploit this redundancy via general-purpose techniques for incremental computation (IC), including memoization, aka function caching (Pugh and Teitelbaum 1989; Acar et al. 2003; Hammer et al. 2014, 2015a). In general, IC offers the promise of extracting responsive *change aware* behavior from simple *change oblivious* algorithms.



In the context of PRY, IC techniques exploit the similarity in states between one time instant and the next, as depicted in the left figure. As the user interacts with the system, its successive states $\mathtt{st}_1$ and $\mathtt{st}_2$ accumulate inputs over *user interaction time*, shown horizontally. Meanwhile, at each point during this interaction, the `compute` algorithm (shown vertically) produces similar outputs, $\mathtt{out}_1$ and $\mathtt{out}_2$. By expressing `compute` as a purely-functional program, it can exploit IC techniques that work behind the scenes to relate the old work to the new work, and avoid recomputing redundant sub-computations.



Furthermore, unlike popular imperative, event-based frameworks for building interactive systems, IC allows programmers to decompose `compute` into sub-computations (shown as C1 and C2) using ordinary *function composition*. Behind the scenes, IC composes the incremental behavior of these functions in a systematic way. Hence, using IC for `compute` promotes greater *compositionality*, without sacrificing performance.

This paper demonstrates the PRY method applied to modeling and implementing IC-Edit, a simple and responsive text editor. First, we present a simple model for defining the IC-Edit's behavior in terms of a simple formal semantics (Section 3). This simple model exhibits the principles of the PRY paradigm: It decomposes conceptually-distinct steps of the behavior's input-to-output pipeline into separate relations, and it decomposes the behavior of the features into separate inference rules, one (or two) per feature.

Based on this formal semantics, a functional programmer can readily implement these relations as pure functions; we implemented them in both OCaml and Rust (Section 5). In particular, these implementations' designs are independent of any underlying framework design for interaction, and they use neither event-driven callbacks nor global state. Like the formal semantics, they use purely-functional data structures and algorithms, representing editable sequences with the *list zipper* pattern, a well-known "functional pearl" (Huet 1997).

Based on this simple model, we develop a responsive model for IC-Edit that improves its performance while preserving both its input-output behavior and compositional design. In particular, we overcome the performance issues of the simple model by employing general-purpose IC to cache and reuse its work over time. In addition, we enhance its zipper-based representation of editable sequences by introducing the *random access zipper*. The random-access zipper is a novel data structure that supplements traditional zippers with efficient random-access refocusing. Though we introduce it here to implement IC-Edit via the PRY methodology, it is general enough to be of independent interest (Section 4). Likewise, we implement the *Adapton approach* for general-purpose IC as a new Rust library, and enhance existing techniques with several new optimizations (Section 5).

We empirically evaluate the simple and responsive implementations of IC-Edit, as well as the proposed IC optimizations (Section 6). Overall, we find that the performance of the simple implementation of IC-Edit degrades over time, as the number of prior user actions grows (viz., the size of the state increases). Under a light workload that consists only of local edits and which eschews non-local navigation, the simple implementation exhibits response time that is linear in the length of its action history; under a heavy workload that interleaves edits at randomized positions, its response time is quadractic in the length of its action history. By contrast, under both workloads the responsive implementation exhibits near-constant response times. After about two thousand actions, it outperforms the simple implementation; at 200k actions, its average response time is still only 8ms.

Following our emprical evaluation, we review related work (Section 7). As we discuss there, we are certainly not the first researchers to propose new methodologies for programming interactive systems. In particular, functional reactive programming (FRP) is similar in its promotion of *compositional design* within existing languages for *functional programming*. As we discuss in Section 7, PRY is distinct in its promotion of a radically simple definition of state as "full input history" whereas FRP is more permissive, allowing systems with more complex definitions of reactive state. In sum, we view FRP as orthogonal (and perhaps complementary) to the proposed PRY methodology.

2. Overview

In this section, we present an overview of applying the PRY methodology to the design and implementation of IC-Edit. We demonstrate an example of interacting with IC-Edit, which informally illustrates how its `compute` behavior determines its output from a history of user input actions. We sketch the *simple model* of IC-Edit, in terms of a formal semantics and corresponding naïve implementation. To address the practical limitations of the simple model, the *responsive model* uses IC techniques and the random access zipper. We illustrate this data structure with an example.

An Example of Interacting with IC-Edit. Figure 1 lists nine input actions (left column), and the ten symbol sequences before and after IC-Edit processes each action (right column). The initial sequence consists of the symbol "z" and "y", an active cursor γ, the symbol "c", an inactive cursor α (circled to emphasize its inactivity), and the symbols "d", and "e". The first action overwrites the "y" symbol

	Action	Symbol sequence
	initial	z y ⌶γ c ⓐ d e
1	ovr L b	z ⌶γ b c ⓐ d e
2	undo	z y ⌶γ c ⓐ d e
3	redo	z ⌶γ b c ⓐ d e
4	rem L	⌶γ b c ⓐ d e
5	ins a L	a ⌶γ b c ⓐ d e
6	move L	⌶γ a b c ⓐ d e
7	join α	a b c ⌶α d e
8	make β	a b c ⓑ ⌶α d e
9	goto 0	⌶α a b c ⓑ d e

Figure 1: An example of interacting with IC-Edit

to the left of the cursor, changing it to a "b" and moving left. The second action is undo, which recovers the initial symbol sequence. Following this undo, the third action is redo, which re-performs the first overwrite action (viz., ovr L b).

Next, actions 4–6 consist of local edits: The forth action removes the "z" symbol to the left of the cursor; this removal action is followed by a left insertion of symbol "a". The sixth action moves the active cursor one text symbol to the left without altering the symbol sequence.

Finally, actions 7–9 demonstrate non-local cursor commands. Action 7 is an example of a cursor join, which eliminates the active cursor from the symbol sequenced (currently γ), and activates the specified cursor, in this case α. Action 8 creates a new cursor β immediately to the left of the current position. Finally, action 9 consists of a goto command, specifying that the active cursor move immediately to the left of symbol position 0, the leftmost position in the sequence.

2.1 PRY Design Phase 1: Simple Model of IC-Edit

The left fragment of Figure 2 illustrates the first phase of applying PRY to the design of IC-Edit. In particular, it yields the simple model of IC-Edit's `compute` behavior in terms of a two-stage pipeline. The first stage processes input actions A by evaluating undo and redo actions; it produces a residual sequence of (non-undo, non-redo) editing commands C. The second stage processes the command sequence C, producing a focused sequence of output symbols, represented as a *symbol zipper* Z.

Recalling the example from Figure 1, the full sequence of actions A consists of the left column (viz., actions 1–9), prepended with an initial action sequence A_0 that determines the initial symbol sequence shown in the first row. After the first stage, the redo of action 3 cancels the undo of action 2, and the residual command sequence C includes of all the remaining commands shown, viz., action 1 followed by 4–9 (and prepended with residual commands from A_0 that produce the first row). Lastly, the focused symbol sequence Z matches that of the final row, after action 9: ⌶α a b c ⓑ d e.

The following judgement makes this pipeline precise:

$$\frac{\gamma \text{ fresh} \quad \langle \epsilon \mid \gamma \mid \epsilon \rangle \vdash \mathsf{rev}(C_1) \Downarrow Z \qquad A \Downarrow \langle C_1 \mid C_2 \rangle}{A \Downarrow Z}\ \textsc{IC-Edit/Pipeline}$$

The single-rule judgement $A \Downarrow Z$ relates the full history of input actions A with the current focused symbol sequence, represented as zipper Z. The rule consists of three premises. The first premise, γ fresh, merely chooses an arbitrary cursor γ not mentioned in action sequence A; it acts as the initial "default" cursor. The next premise uses the first stage's formal semantics to derive a residual sequence of commands C_1. Additionally, it yields a residual undo buffer C_2, which the rule discards. Beginning with the initial cursor γ and an empty symbol zipper, the final premise uses the second stage's formal semantics to derive symbol zipper Z from the residual commands sequence C_1. This premise reverses these commands because the first premise produces them in reverse order, for technical convenience.

Section 3 gives the complete formal semantics of IC-Edit, which consists of five judgement forms and 26 inference rules. Compared with an imperative prototype based on events and callbacks, this model *is simple* because it decouples the semantics of undo and redo actions from the commands that those actions determine, and it models each of the two decoupled stages with a simple set of declarative inference rules. When implemented naïvely, the key drawback of this semantics is its performance. In particular, this simple model exhibits two performance-related issues: First, for each new input action, its two stages repeatedly process (*nearly*) the same sequences of actions and commands. Second, for commands involving non-local cursor

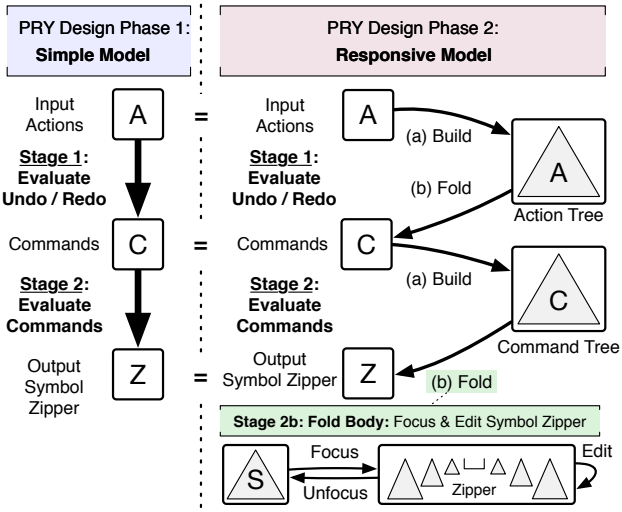


Figure 2: IC-Edit's system design consists of two phases.

movement (e.g., goto and join, shown above), a naïve implementation of this model uses linear search, which does not efficiently scale as the symbol sequence grows.

2.2 PRY Design Phase 2: Responsive Model of IC-Edit

The right fragment of Figure 2 illustrates the second phase of applying PRY to the design of IC-Edit. In particular, by refining the two-stage pipeline of the simple model to address the performance issues discussed above, it yields the responsive model of IC-Edit's `compute` behavior. In Stage 1(a), the responsive model builds a probabilistically-balanced tree consisting of the input action sequence A; in Stage 1(b), the responsive model folds over this tree, producing the residual command sequence C. By combining general-purpose IC techniques with this balanced tree, Stage 1 avoids the redundant work of reprocessing nearly identitical action sequences (Hammer et al. 2014, 2015a).

In Stage 2(a), the responsive model mirrors Stage 1(a), building a balanced tree consisting of the command sequence C. Next, mirroring the Stage 1(b), Stage 2(b) folds this tree, producing the final output, the symbol zipper Z. As with Stage 1, the presence of IC and the balanced tree allow Stage 2 to avoid redundant work over time. However, unlike in Stage 1(b), this last stage requires extra care; in particular, evaluating commands that navigate the active cursor non-locally (e.g., join and goto) are each inefficient in the simple model. To address this issue, Stage 2(b) employs a new data structure and associated algorithms, which we collectively refer to as the random access zipper. At a high level, this purely-functional structure provides logarithmic operations to focus and unfocus the symbol zipper, even when successive focal points are chosen at random.

Editing Sequences via the Random Access Zipper (RAZ). Figure 3 (first image) shows the final symbol sequence from Figure 1, interposed with *names*, shown as numerals 1–6. The two cursors from this sequence, α and β, are circled, indicating that they are inactive. In the example that follows, we process three commands: switch to cursor β, remove the text symbol to the left of cursor β (viz., text symbol c), and switch to cursor α.

First, before making cursor β active, we represent the *unfocused* sequence as a probabilistically-balanced tree, shown in the second image of Figure 3. To construct this tree, we associate each sequence element (cursor, text or name) with a *level*, where levels determine these elements' height in the tree. The levels of cursors and text symbols are zero, indicating that they are the leaves of the tree; we assign the levels of names probabilistically using a negative binomial distribution, which yields a balanced tree, in expectation.[1] The third image in Figure 3 shows the zipper that results from focusing the sequence on cursor β. As can be seen, this structure consists of left and right lists that each contain names and unfocused subtrees from the original balanced tree. (The diagram distinguishes tree and list pointers with distinct arrowheads). The focusing algorithm produces this zipper by descending the balanced tree along the indicated focus path (second image of Figure 3), adding names and subtrees along this path to the left and right lists. Notice that the elements nearest to cursor β consist of the subtrees at the end of this path; in expectation, these lists order subtrees in ascending size.

[1] We implement this choice with hashes, similar to Hammer et al. (2015a), where we count the consecutive zeros in the LSBs of the hash value.

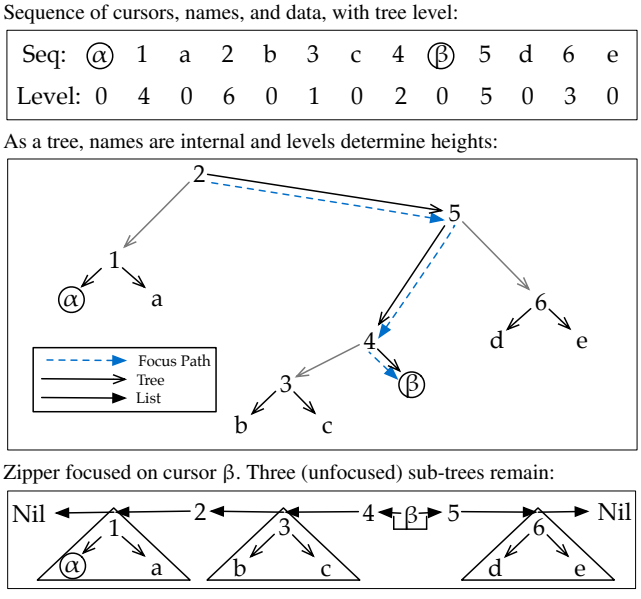


Figure 3: The sequence of symbols and cursors are interposed with names 1–6 (first image); the levels of these names uniquely determine a balanced tree (second image) that permits log-time focusing on cursor β (third image).

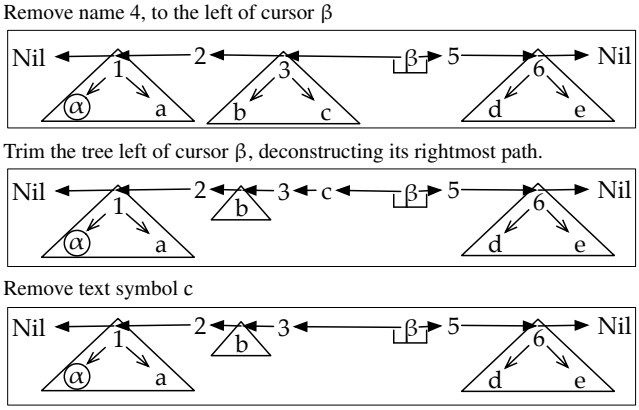


Figure 4: An example of editing a focused zipper: Remove the text symbol c to the left of active cursor β by removing the name 4 (first image), trimming the left tree (second image) and removing symbol c (third image).

Figure 4 shows the three steps for removing the text symbol c to the left of active cursor β; the initial state of this sequence is the final state from Figure 3. First, we remove the name 4 from the left of the active cursor β, making c the next element to the immediate left of the active cursor β (the

top figure). Next, since the text symbol c resides as a leaf in an unfocused tree, we *trim* this left tree by deconstructing its rightmost path (viz., the path to c). Finally, with symbol c exposed in the left list, we remove it (third image).

Beginning with the final state of Figure 4 and active cursor β, Figure 5 illustrates the process of unfocusing the sequence, and then refocusing on cursor α. First, we add active cursor β to the left list, storing its position in the sequence (second image). Next, we build trees from the left and right lists as follows: For each list, we fold its elements and trees, appending them into balanced trees; as with the initial tree, we use the levels of names to determine the height of internal nodes (third image). Having created two balanced trees from the left and right lists, we append them along their rightmost and leftmost paths, respectively; again, the append path compares the levels of names to determine the final appended tree (forth image). Finally, as in Figure 3, we descend the focus path to the desired cursor, this time cursor α. As before, this path induces left and right lists that consist of names and unfocused subtree (fifth image).

3. The Formal semantics of IC-Edit

In this section, we present a simple and complete formal semantics for IC-Edit. In particular, we formalizes a precise language semantics for end-user editing of sequential data; though we think of the atoms of these sequences as text symbols, this design easily generalizes to arbitrary sequential data. The language semantics we present underpins both the simple and responsive implementations of IC-Edit.

The formal language of IC-Edit centers on imperative commands that modify the text buffer and manage the pool of available cursors. Formally, a user action $\mathfrak{a}$ consists of an editing command c, or the actions that undo and re-do past commands (undo and redo):

$$\begin{array}{rrcl} \text{Action} & \mathfrak{a} & ::= & \mathsf{cmd}\ \mathsf{c} \mid \mathsf{undo} \mid \mathsf{redo} \\ \text{Command} & \mathsf{c} & ::= & \mathsf{ins}\ \mathsf{d}\ \mathsf{t} \mid \mathsf{rm}\ \mathsf{d} \mid \mathsf{ovr}\ \mathsf{d}\ \mathsf{t} \mid \mathsf{mv}\ \mathsf{d} \mid \mathsf{goto}\ \mathfrak{n} \\ & & \mid & \mathsf{mk}\ \alpha \mid \mathsf{sw}\ \alpha \mid \mathsf{jmp}\ \alpha \mid \mathsf{join}\ \alpha \\ \text{Direction} & \mathsf{d} & ::= & \mathsf{L} \mid \mathsf{R} \end{array}$$

Specifically, an editing command c consists of inserting, removing, or overwriting text symbols (ins, rm, ovr); moving the active cursor within the text (mv and goto); making new cursors (mk); and moving among existing cursors by switching between, jumping to, or joining with them (sw, jmp, join). We use d to range over two possible directions (viz., left L and right R), t to range over an unspecified set of text symbols, and α to range over an unspecified set of cursor symbols.

At each state during user interaction, the semantics models the text buffer as a *symbol sequence* S consisting of cursor and textual symbols. The semantics of commands distinguishes a special *active cursor*, which in turn induces a *symbol zipper* $\langle S_1 \mid \alpha \mid S_2 \rangle$ that consists of symbol sequences to the left (S_1) and right (S_2) of the active cursor α:

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

Using these structures, Figures 6 and 7 give zipper-transforming semantics for single commands c and command sequences C, respectively, by defining the judgement forms $Z_1 \vdash c \longrightarrow Z_2$ and $Z_1 \vdash C \Downarrow Z_2$. The first judgement is read as "the command c transforms initial zipper Z_1 into final zipper Z_2", and the second is read similarly: "the command sequence C transforms initial zipper Z_1 into final zipper Z_2". These semantics are directly inspired by the *zipper pattern*, which provides a general approach for describing purely functional data structures that undergo small changes. First proposed for n-ary trees by Huet, the zipper pattern has

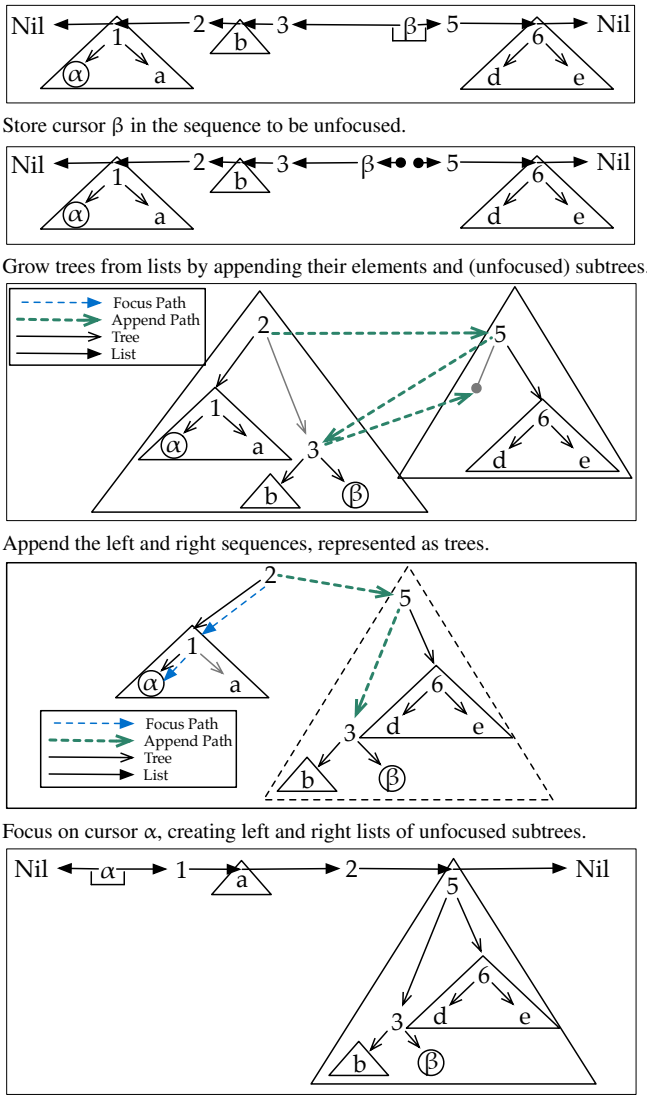


Figure 5: Switch from cursor β to α by unfocusing and refocusing.

$\boxed{Z_1 \vdash C \Downarrow Z_2}$ *Under* Z_1*, performing commands* C *yields* Z_2

$$\frac{}{Z \vdash \epsilon \Downarrow Z}\ \textsf{Cs\_nil} \qquad \frac{Z_1 \vdash c \longrightarrow Z_2 \quad Z_2 \vdash C \Downarrow Z_3}{Z_1 \vdash c :: C \Downarrow Z_3}\ \textsf{Cs\_cons}$$

Figure 6: Stage 2 of Pipeline: Commands to Symbols.

$\boxed{Z_1 \vdash c \longrightarrow Z_2}$ *Under* Z_1*, command* c *yields zipper* Z_2

$$\frac{}{\langle S_1 \mid \alpha \mid S_2\rangle \vdash \textsf{ins t L} \longrightarrow \langle S_1 :: t \mid \alpha \mid S_2\rangle}\ \textsf{C\_insertL1}$$

$$\frac{\langle S_1 \mid \alpha \mid S_2\rangle \vdash \textsf{ins t L} \longrightarrow \langle S_1' \mid \alpha \mid S_2'\rangle}{\langle S_1 :: \beta \mid \alpha \mid S_2\rangle \vdash \textsf{ins t L} \longrightarrow \langle S_1' :: \beta \mid \alpha \mid S_2'\rangle}\ \textsf{C\_insertL2}$$

$$\frac{\textsf{rev}(Z) \vdash \textsf{ins t L} \longrightarrow \textsf{rev}(Z')}{Z \vdash \textsf{ins t R} \longrightarrow Z'}\ \textsf{C\_insertR}$$

$$\frac{}{\langle S_1 :: t \mid \alpha \mid S_2\rangle \vdash \textsf{rm L} \longrightarrow \langle S_1 \mid \alpha \mid S_2\rangle}\ \textsf{C\_removeL1}$$

$$\frac{\langle S_1 \mid \alpha \mid S_2\rangle \vdash \textsf{rm L} \longrightarrow \langle S_1' \mid \alpha \mid S_2'\rangle}{\langle S_1 :: \beta \mid \alpha \mid S_2\rangle \vdash \textsf{rm L} \longrightarrow \langle S_1' :: \beta \mid \alpha \mid S_2'\rangle}\ \textsf{C\_removeL2}$$

$$\frac{\textsf{rev}(Z) \vdash \textsf{rm L} \longrightarrow \textsf{rev}(Z')}{Z \vdash \textsf{rm R} \longrightarrow Z'}\ \textsf{C\_removeR}$$

$$\frac{}{\langle S_1 :: t \mid \alpha \mid S_2\rangle \vdash \textsf{mv L} \longrightarrow \langle S_1 \mid \alpha \mid t :: S_2\rangle}\ \textsf{C\_moveL1}$$

$$\frac{\langle S_1 \mid \alpha \mid \beta :: S_2\rangle \vdash \textsf{mv L} \longrightarrow Z}{\langle S_1 :: \beta \mid \alpha \mid S_2\rangle \vdash \textsf{mv L} \longrightarrow Z}\ \textsf{C\_moveL2}$$

$$\frac{\textsf{rev}(Z) \vdash \textsf{mv L} \longrightarrow \textsf{rev}(Z')}{Z \vdash \textsf{mv R} \longrightarrow Z'}\ \textsf{C\_moveR}$$

$$\frac{Z_1 \vdash \textsf{rm L} \longrightarrow Z_2 \quad Z_2 \vdash \textsf{ins t R} \longrightarrow Z_3}{Z_1 \vdash \textsf{ovr t L} \longrightarrow Z_3}\ \textsf{C\_overwriteL}$$

$$\frac{Z_1 \vdash \textsf{rm R} \longrightarrow Z_2 \quad Z_2 \vdash \textsf{ins t L} \longrightarrow Z_3}{Z_1 \vdash \textsf{ovr t R} \longrightarrow Z_3}\ \textsf{C\_overwriteR}$$

$$\frac{\gamma\ \textsf{fresh} \quad \langle S_1 :: \alpha \mid \gamma \mid S_2\rangle \leftrightarrow \langle S_1' :: \beta \mid \gamma \mid S_2'\rangle}{\langle S_1 \mid \alpha \mid S_2\rangle \vdash \textsf{sw}\ \beta \longrightarrow \langle S_1' \mid \beta \mid S_2'\rangle}\ \textsf{C\_switchto}$$

$$\frac{\langle S_1 \mid \alpha \mid S_2\rangle \leftrightarrow \langle S_1' :: \beta \mid \alpha \mid S_2'\rangle}{\langle S_1 \mid \alpha \mid S_2\rangle \vdash \textsf{jmp}\ \beta \longrightarrow \langle S_1' :: \beta \mid \alpha \mid S_2'\rangle}\ \textsf{C\_jumpto}$$

$$\frac{\langle S_1 \mid \alpha \mid S_2\rangle \leftrightarrow \langle S_1' :: \beta \mid \alpha \mid S_2'\rangle}{\langle S_1 \mid \alpha \mid S_2\rangle \vdash \textsf{join}\ \beta \longrightarrow \langle S_1' \mid \beta \mid S_2'\rangle}\ \textsf{C\_join}$$

$$\frac{\langle S_1 \mid \alpha \mid S_2\rangle \leftrightarrow \langle S_1' \mid \alpha \mid S_2'\rangle \quad |S_1'|_{\textsf{text}} = n}{\langle S_1 \mid \alpha \mid S_2\rangle \vdash \textsf{goto}\ n \longrightarrow \langle S_1' \mid \alpha \mid S_2'\rangle}\ \textsf{C\_goto}$$

$$\frac{}{\langle S_1 \mid \alpha \mid S_2\rangle \vdash \textsf{mk}\ \beta \longrightarrow \langle S_1 :: \beta \mid \alpha \mid S_2\rangle}\ \textsf{C\_mk}$$

Figure 7: Semantics of IC-Edit's commands.

$\boxed{Z_1 \leftrightarrow Z_2}$ *Zipper* Z_1 *refocuses to* Z_2 *in zero or more steps.*

$$\frac{}{Z \leftrightarrow Z}\ \textsf{Refl} \qquad \frac{\langle S_1 \mid \alpha \mid s :: S_2\rangle \leftrightarrow Z}{\langle S_1 :: s \mid \alpha \mid S_2\rangle \leftrightarrow Z}\ \textsf{L} \qquad \frac{\langle S_1 :: s \mid \alpha \mid S_2\rangle \leftrightarrow Z}{\langle S_1 \mid \alpha \mid s :: S_2\rangle \leftrightarrow Z}\ \textsf{R}$$

Figure 8: Non-determinisitic refocusing of symbol cursor.

since been adapted and generalized extensively (Huet 1997; Abbott et al. 2004; Ramsey and Dias 2006).

The C_insertL1 rule gives a prototypical example of local editing with a zipper. The command inserts the text symbol t to the left of the active cursor; in the righthand side, the left symbol sequence S_1 is merely extended with t, while the right sequence S_2 remains unchanged. In this rule, and below, we use "snoc" notation for sequences that grow from left to right, viz., we write $S :: t$ instead of the more common "cons" notation $t :: S$; this is merely a notational convention to increase readability in the rules.

The C_insertL2 rule addresses the case when the symbol immediately to the left of the active cursor is not textual, but instead is another (inactive) cursor β. We first experimented with a semantics where this case was not distinguished, but we found that it lead to somewhat unintuitive behavior—amongst a group of consecutive cursors, the active cursor would insert text between it and the adjacent (inactive) cursors. By contrast, the design of the C_insertL2 rule inserts text to the left of any cursors that are immediately adjacent to the active cursor. Finally, C_insertR mirrors the left-acting insertion rules by using an auxiliary function on symbol zippers rev; this function merely exchanges the left and right symbol sequences of the zipper; i.e., $\textsf{rev}(\langle S_1 \mid \alpha \mid S_2\rangle) = \langle S_2 \mid \alpha \mid S_1\rangle$. The rules for removal and local movement (rm and mv) employ patterns that are analogous to those of the ins command. Finally, the rules for the overwrite command (ovr) pair a removal with an insertion.

Unike the local editing commands, the non-local editing commands each manipulate the placement of the active cursor in a non-local fashion. As an example, the C_join rule gives the semantics of the join command. The rule uses a single premise that non-deterministically refocuses near the target cursor β. In general, we read $Z_1 \leftrightarrow Z_2$ as "symbol zipper Z_1 refocuses to Z_2"; the general rules for this refocusing judgement are given in Figure 8. The judgement preserves the order of symbols in the zipper, but non-deterministically moves the focus to another position in the sequence of symbols. In the conclusion of the C_join rule, the original active cursor α is *eliminated*, and replaced with the target cursor β. The other cursor movement rules (viz., switch, jump and goto) each use the refocusing judgement in their premises.

Evaluating undo and redo actions. Before evaluating a residual sequence of commands C, the first stage of IC-Edit's compute pipeline evaluates undo and redo actions. We formalize this aspect of editing by employing a zipper for

$\boxed{A \Downarrow Z_C}$ *Evaluating actions A yields command zipper* Z_C

$$\frac{}{\epsilon \Downarrow \langle \epsilon \mid \epsilon \rangle}\ \text{A\_nil} \qquad \frac{A \Downarrow \langle C_1 :: c \mid C_2 \rangle}{A :: \text{undo} \Downarrow \langle C_1 \mid c :: C_2 \rangle}\ \text{A\_undo}$$

$$\frac{A \Downarrow \langle C_1 \mid c :: C_2 \rangle}{A :: \text{redo} \Downarrow \langle C_1 :: c \mid C_2 \rangle}\ \text{A\_redo} \qquad \frac{A \Downarrow \langle C_1 \mid C_2 \rangle}{A :: \text{cmd } c \Downarrow \langle C_1 :: c \mid \epsilon \rangle}\ \text{A\_cmd}$$

Figure 9: Stage 1 of Pipeline: Actions to Commands.

command sequences: $Z_C ::= \langle C_1 \mid C_2 \rangle$. Figure 9 defines the judgement $A \Downarrow Z_C$, which formalizes the semantics of evaluating an action sequence A to produce a residual command sequence C_1 and undo buffer C_2. We read this judgement as "performing action sequence A yields command history C_1, and undo buffer C_2." In particular, the EA_undo rule evaluates an undo action by moving the last command c from the command sequence to the undo buffer, and the EA_redo rule does the reverse. Notably, the EA_cmd rule evaluates a command by appending it to the residual command sequence, and by *clearing* the resulting undo buffer. This semantics faithfully matches that of the vast majority of interactive applications with undo features. For example, it is common in end-user software for the two sequences of actions to result in the same state:

Sequence 1 : cmd c_1, cmd c_2, undo, cmd c_3, redo
Sequence 2 : cmd c_1, cmd c_3

In particular, while the undo in the first sequence undoes command c_2, the final redo in this sequence has no effect: The preceeding command c_3 clears the undo buffer before it can redo the undone command c_2.

Performance challenges in simple model. The simple model described above has two key performance challenges, which each stem from the simplicity of its semantics, and its adherence to the PRY methodology.

First, for each new user action, the simple model *fully repeats* the two pipeline phases described above. Repeating phase one consists of recomputing the reversed command buffer $\text{rev}(C_1)$ for the *entire history* of user actions A. Repeating phase two consists of recomputing the final symbol zipper Z_2 for the *entire sequence* of commands in this buffer, $\text{rev}(C_1)$. The responsive model overcomes this first challenge without changing the approach conceptually; instead, it employs incremental computation to avoid performing any redundant prefixes of these two phases.

Next, in addition to being repetitive, the second phase of the simple model's pipeline evaluates commands that require zipper refocusing by using a naïve search strategy (e.g., a linear search implementation of judgement $Z_1 \leftrightarrow Z_2$), rather than use a more efficient approach with greater complexity. Again, the responsive model overcomes this second challenge without changing the approach conceptually: It still uses a zipper-based representation for the symbol sequence. However, to support randomized access, it enhances this representation with additional structure that permits efficient (log-time) unfocusing and focusing operations.

4. Random Access Zipper

The Random Access Zipper (RAZ) is critical to IC-Edit's responsive implementation. Unlike a traditional list zipper (e.g., those in Section 3), a RAZ efficiently supports non-local movement in addition to simple local edits at its current focal position. In this sense, the RAZ offers the combined benefits of ordinary list zippers and purely-functional arrays.

As an example, consider the following loop of edits:

```
let rec loop edits seq = match edits with
| [] → seq
| (pos, edit) :: edits →
   let z1 = focus seq pos in        O(log |seq|)
   let z2 = do_edit edit z1 in      O(1)
   let s2 = unfocus z2 in           O(log² |seq|)
   loop edits s2
```

The loop processes a list of `edits`, whose interpretation is given by a zipper-transforming function `do_edit` with type:

```
do_edit : edit → zipper → zipper
```

This function assumes that the zipper is focused on the desired position of the edit, `pos` in the code above. Though not specified above, the representation of `pos` may either be a global offset (such as a line number or character count), or a symbolic, unique cursor identity, as introduced in earlier sections of this paper.

To focus and unfocus the RAZ sequence, the loop uses the two RAZ operations `focus` and `unfocus`, respectively. As illustrated in the example of Section 2.2, the `focus` operation transforms the sequence from a canonical, probabilistically-balanced tree to a list of trees, whose representation admits local editing, a la traditional list zippers. After one or more local edits, the `unfocus` operation inverts this step, recovering an edited balanced tree. Section 2.2 depicts an example of using these operations, and in this section, we present a general approach in the form of complete code listings and performance bounds. In particular, we show that `focus` and `unfocus` operations used above each run in sublinear time, viz., $O(\log(\cdot))$ and $O(\log^2(\cdot))$ time, respectively. As a result, the loop above runs in time $O(|\texttt{edits}| \log^2(|\texttt{edits}| + |\text{seq}|))$.

The `zipper` datatype captures the RAZ's structure:

```
type 'a tree   = Nil | Leaf of 'a
               | Bin of name option * 'a tree * 'a tree
type 'a tlist  = Nil | Cons of 'a * 'a tlist
               | Name of name * 'a tlist
               | Tree of 'a tree * 'a tlist
type 'a zipper = 'a tlist * 'a tlist
```

This structure is stratified into three types: the `tree` type consists of (unfocused) binary trees, where leaves hold data and internal binary nodes hold an optional unique `name`; the `tlist` type consists of ordinary list structure, plus two

```
let focus : cur → sym tree → (sym zipper) option
= fun cur → let rec loop
  : sym tree → sym zipper → (sym zipper) option
  = fun tree z → match tree with
    | Nil      → None
    | Leaf(l) → if l = Cur(cur) then Some(z)
                else None
    | Bin(n, l, r) →
       if      find cur (tree_info l).curs
       then    loop l (insert_tree n r R z)
       else if find cur (tree_info r).curs
       then    loop r (insert_tree n l L z)
       else    None
  in fun tree → loop tree (Nil, Nil)
```

Figure 10: Focus a `tree` on a cursor `cur`, creating a zipper

```
let unfocus : 'a zipper → 'a tree =
  fun (left, right) →
    let ltree = grow L (next_tree left ) in
    let rtree = grow R (next_tree right) in
    append ltree rtree
```

Figure 11: Unfocus a `zipper`, creating a `tree`

`Cons`-like constructors that hold `names` and `trees` instead of ordinary data; finally, a (focused) `zipper` consists of a left and right `tlist`.

Focusing the random-access zipper. The `focus` operation in Figure 10 transforms an unfocused tree to a focused zipper. Given a cursor `cur` symbol, a unique position in the tree, and an $O(1)$-time `find` operation on sub-trees, the inner `loop` recursively walks through one path of `Bin` nodes until it finds the desired `Leaf` symbol. At each step of this walk, the `insert_tree` operation accumulates un-walked subtrees in the zipper `z`; in the base case, `focus` returns this accumulated `z`. The walk is guided by `find`, which indicates which subtree contains the desired cursor symbol. In Section 5, we use the memoization afforded by IC to implement `find` in expected constant time; when IC is absent, one can augment the `Bin` case of the tree type with satellite information to provide efficient indexing, viz., either the size of the subtree, or a ($O(1)$-sized) set of cursors in the subtree, or both. Under the following conditions, `focus` is efficient, running in logarithmic time for balanced trees:

Proposition 4.1. *Given a tree* `t` *of depth* `d`*, and an* $O(1)$*-time implementation of* `find`*, the operation* `focus cur t` *runs in* $O(d)$ *time.*

Unfocusing the random-access zipper. The `unfocus` operation in Figure 11 transforms a focused `zipper` to an unfocused `tree`. It uses auxiliary operation `grow` to construct trees for the `left` and `right` `tlist` sequences that comprise the focused RAZ. The first call to `grow` and its recursive computation each use `next_tree` to extract successive `trees` from a given `tlist`. The final step of `unfocus` consists of appending the left and right trees, `ltree` and `rtree`, respectively. Under the conditions stated below, `unfocus` is efficient, running in polylogarithmic time for balanced trees with logarithmic depth:

```
let next_tree : 'a tlist → ('a tree * 'a tlist)
  = fun tlist → match tlist with
           | Nil       → (Nil, Nil)
           | Cons(s,r) → (Leaf(s),r)
           | Tree(t,r) → (t,r)
           | Name(n,r) → (Bin(Some(n),Nil,Nil),r)
```

Figure 12: From a `tlist`, get the next structure as a `tree`

```
let grow : dir → ('a tree * 'a tlist) → 'a tree
= fun dir →
  let append' : 'a tree → 'a tree → 'a tree =
    match dir with
    | L → fun t1 t2 → append t1 t2
    | R → fun t1 t2 → append t2 t1
  in
  let rec loop (t1, rest) =
    if rest = Nil then t1 else
      let (t2, rest) = next_tree rest in
      loop (append' t1 t2, rest)
  in loop
```

Figure 13: Grow a single balanced tree from a `tlist`

Proposition 4.2. *Given a tree* `t` *of depth* `d`*, peforming* `unfocus (unwrap (focus t cur))` *requires* $O(d)$ *time.*

We sketch the reasoning for this claim, deferring details about the auxiliary functions `grow` and `next_tree` to the discussion below. As stated above, the operation `focus t cur` runs in $O(d)$ time; we further observe that `focus` produces a zipper with left and right lists of length $O(d)$. Assuming `cur` is present, the `unwrap` operation eliminates the `option` type in constant time (and raises an exception otherwise). Likewise, `next_tree` also runs in constant time. Next, the `unfocus` operation uses `grow` to produce left and right trees in $O(d)$ time. In general, `grow` makes `d` calls to `append`, combining trees of height approaching `d`, requiring $O(d^2)$ time. However, since these trees were placed *in order* by `focus`, each `append` here only takes constant time. Finally, it `appends` these trees in $O(d)$ time. None these steps dominate asymptotically, so the composed operations run in $O(d)$ time.

Iterating, growing and appending trees. The `unfocus` operation above consists of calls to the auxiliary operations `next_tree`, `grow`, and `append`. Figure 12 lists `next_tree`, which transforms a `tlist` to a pair consisting of a `tree` and a residual `tlist`; conceptually, it extracts the next tree, leaf data or binary node name as `tree` structure. In turn, the `grow` operation in Figure 13 loops over successive trees,

```
let rec append : 'a tree → 'a tree → 'a tree =
  fun t1 t2 → match t1, t2 with
  | Nil, _ → t2 | _, Nil → t1
  | Leaf(_), Leaf(_)     → Bin(None, t1, t2)
  | Leaf(_), Bin(n,l,r) → Bin(n, append t1 l, r)
  | Bin(n,l,r), Leaf(_) → Bin(n, l, append r t2)
  | Bin(n1,t1l,t1r), Bin(n2,t2l,t2r) →
     if level n1 > level n2
     then Bin(n1, t1l, append t1r t2)
     else Bin(n2, append t1 t2l, t2r)
```

Figure 14: Append sequences represented as balanced trees

each extracted by `next_tree`, and it accumulates them in a loop, combining them with `append`. The `dir` parameter determines whether the accumulated tree grows from left-to-right (`L` case), or right-to-left (`R` case); the internal function `append'` specializes the general `append` operation to this choice. When the `tlist` is `Nil`, the `loop` within `grow` completes, and yeilds the accumulated tree `t1`.

The `append` operation in Figure 14 produces a tree whose leaves and internal names consist of the leaves and names of the two input trees, in order. That is, an in-order traversal of the tree result of `append t1 t2` first visits the names and leaves of tree `t1`, followed by the leaves and names of tree `t2`. The algorithm works by traversing a path in each its two tree arguments, and producing an appended tree with the aforementioned in-order traversal property. In the `Bin` node case, the computation chooses between descending into the sub-structure of argument `t1` or argument `t2` by comparing their names' levels and by choosing the tree named with the higher level. As depicted in the example in Figure 5 (from Section 2.2), this choice preserves the property that `Bin` nodes with higher levels remain higher in the resulting tree. When these levels are chosen from a negative binomial distribution, this choice preserves the resulting tree's balance, in expectation. Below, we discuss further meta-properties of this algorithm, and compare it to prior work.

Trimming an inner tree The `trim` operation in Figure 15 prepares a `tlist` for edits in the given direction `dir`. It returns the `tlist` unchanged if it does not contain a tree. If the `tlist` does contain a tree, `trim` deconstructs it recursively. Each recursive call eliminates a `Bin` node, pushing the first branch, name, and second branch into the `tlist`. The recursion ends when `trim` reaches a `Leaf` and pushes it into the `tlist` as a `Cons`.

The `trim` operation works most efficiently immediately after a balanced tree is focused into a zipper. The cursor will be surrounded by leaves or small subtrees, which can be trimmed in constant time. If the cursor moves far through the zipper, it can encounter a node from high in the original tree, containing a significant proportion of the total data count.

These facts suggest the following propositions:

```
let trim : dir → 'a tlist → 'a tlist =
 fun dir → let rec loop tlist = match tlist with
 | Nil | Cons(_,_) | Name(_,_) → tlist
 | Tree(tree, rest) →
  match tree with
  | Nil → rest
  | Leaf(l) → Cons(l,rest)
  | Bin(n,l,r) →
    match dir, n with
    | L, None → loop (Tree(l,Tree(r,rest)))
    | R, None → loop (Tree(r,Tree(r,rest)))
    | L, Some(n) → loop (Tree(l,Name(n,Tree(r,rest))))
    | R, Some(n) → loop (Tree(r,Name(n,Tree(r,rest))))
in loop
```

Figure 15: Trim a `tree`, on direction `dir`, creating a `tlist`

Proposition 4.3. *Given a direction* `d`*, a cursor* `c`*, a tree* `t` *of size* n*, and a* `tlist` `l` *from one side of a zipper created by* `focus t c`*,* `trim d l` *runs in* $O(1)$ *time.*

Proposition 4.4. *Given a direction* `d`*, a cursor* `c`*, a tree* `t` *of size* n*, and a* `tlist` `l` *from one side of a zipper created by* `focus t c`*, a sequence of* k *calls to* `trim d l` *composed with* `move d` *runs in* $O(k \log n)$ *time.*

Discussion of tree balancing via `append`**.** The `append` algorithm presented here is inspired by that of Pugh and Teitelbaum (1989). A key difference is our introduction of names, which address the case when the sequenced data elements are indistinguishable (e.g., to represent a text buffer with a single character repeated an arbitrary number of times). In particular, prior work lacked names, and instead assumed *unique data elements* determine a balanced tree.

As with Pugh and Teitelbaum's algorithm, our version of `append` is particularly well-suited to memoization via IC, since it yeilds a *canonical tree* for a given resulting sequence. In other words, `append` commutes with (ordinary) sequence concatenation, and is associative:

Proposition 4.5. *Given three trees* `t1`*,* `t2` *and* `t3`*, if we have that* `t = append (append t1 t2) t3` *and* `t' = append t1 (append t2 t3)`*, then* `t = t'`*.*

In the next section, we use *hash-consing* to exploit this fact, and avoid storing redundant copies of the same subsequences. As discussed below, names serve a second role: they act as cues to the IC technique of *nominal memoization*.

5. Implementation

We implement the simple and responsive models of IC-Edit, including random access zippers. We also implement Adapton in Rust, enhancing prior work with new optimizations.

5.1 Simple and Responsive implementations of IC-Edit

As prescribed by PRY, the first phase of IC-Edit design consists of creating the formal semantics presented in Section 3, and a simple, executable implementation of this semantics. We create two such prototypes: One in OCaml, a language

that we know well, and later, one in Rust, a language that we are still learning as developers. Our OCaml implementation totals 400 lines of (commented/documented) code, which consists of types and helper functions (~80 lines), the formal semantics itself (~100 lines), random action generation (~45 lines), graphical display (~75 lines) and translation of system-level input events into IC-Edit's language of user actions (~100 lines).

Informed by this initial design and development experience, we created a second simple implementation in Rust totaling 1700 lines, with similar percentages of lines devoted to the tasks of defining types, generating random actions, interacting with system-level graphics and keyboard input. Compared with OCaml, Rust generally consumed more lines for the same conceptual step, even though the complexity of this code is comparable to the OCaml equivalent; its a more verbose language which which we are less familar. Nevertheless, the core aspects of the formal semantics only totaled ~275 lines of Rust: Stage 1 consists of ~35 lines, and Stage 2 consists of ~240 lines; each are purely functional.

Finally, based on the simple implementation in Rust, we created the responsive model and implementation in Rust. To do so, we used implementations of Adapton and the RAZ, each described below. Beyond these (general, reusable) components, the remaining implementation of the responsive model of IC-Edit consists of the stages illustrated in Section 2: Stage 1(a) builds a probabilistically-balanced tree using a single RAZ library call; Stage 1(b) folds over this tree, producing the residual command sequence (~40 lines); Stage 2(a) builds a balanced tree of these commands with a single call; Stage 2(b) folds this tree, producing the final output, the symbol zipper (~110 lines).

Finally, the responsive pipeline ends with graphics code that is common to the responsive and simple implementations (~200 lines). Likewise, both implementations share the start of the pipline, which processes keyboard events from the system (~400 lines), and which generates random actions for regression and performance tests (~225 lines). In addition comparing the two implementations empirically (Section 6), we used regression tests throughout our development that validate the correctness of the responsive implementation against the input/output behavior of the simple version.

5.2 Random Access Zipper in Rust

In Section 4, we present the RAZ data structure and algorithms as OCaml code listings. Their implementation in Rust using the Adapton library (below) is straightforward. However, we note two critical details not captured in those listings: First, we compute `tree_info` as a simple bottom-up tree traversal that returns the size of each sub-tree and the list of cursors that it contains. By memoizing this computation with Adapton, we avoid manually maintaining this satellite data, but still match the O(1)-time and -space performance of doing so. Second, since Adapton requires O(1)-time operations that compare and hash data structures, we use Adapton to hash-cons the trees of the RAZ; further, the *sparse memoization* optimization mentioned below uses the presence or absence of names in the RAZ sequence as cues to control the granularity of this hash-consing.

5.3 Adapton in Rust, with Optimizations for PRY

We implement both Adapton (Hammer et al. 2014) and Nominal Adapton (Hammer et al. 2015a) as a single unified library in the Rust programming language (Matsakis and Klock II 2014). As in prior work, the key abstractions that define this library interface are first-class *name* values, named *cells* that hold changing data, and named *thunks* whose results the library caches and reuses. The original work on Adapton used a "structural" approach for identifying cells and thunks—i.e., it identified these structures by their content, and it automatically hash-cons'd redundant copies of cells and thunks. Subsequent work advanced *nominal matching* as a way to systematically use names to *overwrite* cached data and computations, which can often increase incremental reuse and reduce space overhead. Compared with these earlier prototypes (in OCaml), our Rust implementation makes additional advances relevant to the application of PRY to IC-Edit.

Mixed-use of Structural and Nominal Adapton. We observe that the IC-Edit `compute` pipeline illustrated in Section 2 benefits from a *mixture* of nominal and structural Adapton variants. More generally, our library offers simple annotations to mix structural and nominal computations within a unified Adapton engine. We used these annotations in IC-Edit as follows: We annotated the fold body of stage 2(b) as structural, since it reprocesses similiarly-structured RAZ instances during the fold over the command sequence; this annotation leverages structural hash-consing in the body of this fold. Meanwhile, stages 1(a), 1(b) and 2(a) each use Nominal Adapton to eliminate some space overhead of the structural approach. Section 6 compares this *mixed* approach to a variant that does not use nominal matching, showing a modest reduction in space usage for the new, mixed variant.

Dynamic versus Stable Adapton Nodes. Prior implementations of Adapton employ heavyweight dependency graphs to relate cached results to the computations and data on which they depend, which generally change dynamically. Meanwhile, due to the mixed regions described above, the pure (but memoized) computations of IC-Edit's stage 2(b) only depend on immutable hash-cons'd cells, and do not require a full-blown dependency graph, since their results will not change dynamically. Our implementation of Adapton dynamically discovers and exploits the presence of such *stable* computations by eliding dependency edges for any sub-computation that only depends on immutable data, or other stable sub-computations. As Section 6 shows, this optimization significantly improves the responsiveness of IC-Edit, as well as its memory usage.

Sparse versus Dense Memoization. Prior implementations of Adapton employed data structures to represent lists and trees that change over time. In so doing, they tracked *every pointer* of these structures, and cached *every recursive call* that computed over them. In the data structures that implement IC-Edit (including the random access zipper), we experiment with *sparsely*-tracked structures and computations. To do so, we use the presence or absence of first-class names to indicate that a pointer or recursive call should be tracked (when it is "named"), or not (when it lacks an associated name). In Section 6, modest levels of "sparseness" show notable reductions in time and space overhead.

Customized memory management. Finally, unlike prior work, our Rust implementation of Adapton lacks a runtime system providing general-purpose, traversal-based garbage collection. (By contrast, the OCaml runtime system from prior work uses a combination of copy and mark-sweep techniques to automatically manage memory). As with Rust programs generally, our implementation of Adapton and of IC-Edit releases "stack-owned" memory as stack frames are popped. However, we currently *do not* evict cells or thunks from the Adapton library's heap-allocated memoization cache. Instead, Section 6 experiments with an extreme approach that defers all cache management to the operating system's virtual memory system, which uses swap space under high memory load. As we discuss there, our current performance results show that this crude form of "cache eviction" actually works; further, it suggests that future work can systemize preemptive cache eviction within the Adapton library, eliding the need for swap space.

6. Evaluation

This section evaluates IC-Edit's performance, and compares our simple implementation with our responsive implementation. We find that the simple version requires quadratic time to process a list of actions when they include non-local edits, or linear time with only local edits. By contrast, the responsive version requires sub-linear time, performing better after only 1-2k actions, when the average time is under 1ms; overall, we measure an average response time of under 8ms.

Experimental Setup. Our experiments measure the response time required to update the text buffer for each edit in a randomized distribution of user actions, described below. We also measure the current memory usage every 11 edits, as reported by the UNIX tool ps; notably, it reports *resident set size* (RSS), viz., the RAM used by the IC-Edit process, ignoring swap space.

Our experiments measure two modes of interaction, each representing a different distribution of actions, illustrated in Figure 16. The distribution of local actions lacks commands that navigate the cursor non-locally; it represents a "light workload". By contrast, the distribution of global actions simulates concurrent editing with multiple users. It consists

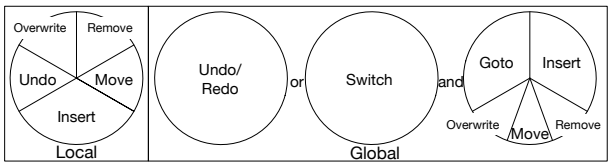


Figure 16: Random action distributions: Local vs global.

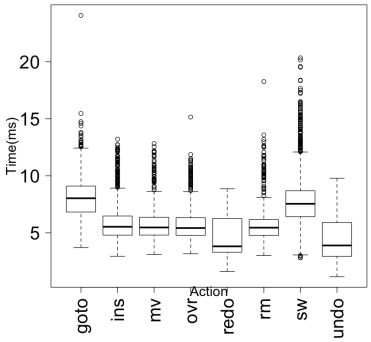
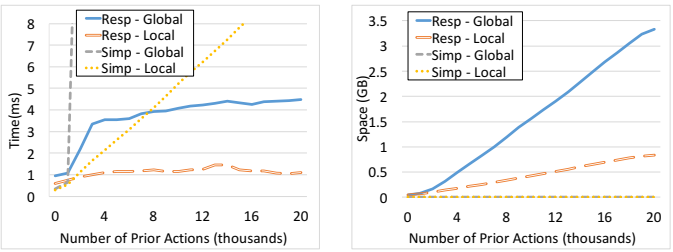


Figure 17: Responsive version at 190k-200k actions

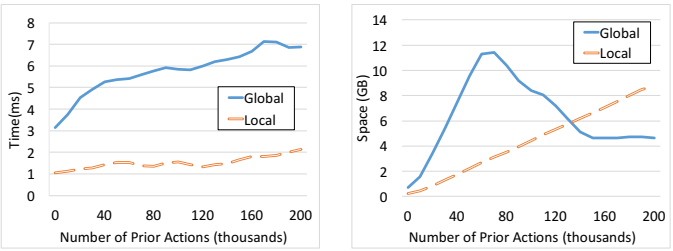
of setup actions that create 32 cursors interposed with 60 padding characters; after this setup, the distribution generates random actions as follows: it either generates a sequence of undos and redos (15% of total actions) or a pair of actions that consist of one cursor switch (to the next cursor among the 32-cursor sequence), and one random editing action, as shown at the right of Figure 16. Since this global-edit distribution consists of both switch and goto actions, it represents a "heavy workload" for IC-Edit. In both distributions, we generate undo and redo actions by generating a sequence of 10 undo actions, and then with 0.5 probabilty, 5 redo actions.

We evaluate the three Adapton optimizations from Section 5 by evaluating their efficacy on the responsive implementation of IC-Edit. First, we compare structural memoization with a mixed-use of structural and nominal memoization. Next, we compare (ordinary) dynamic dependency nodes with the optimization that employs (cheaper) stable nodes. Finally, we compare dense dependence tracking with versions that use the (cheaper) sparse optimization; we measure four settings in total: Dense and Sparse-$\{2–4\}$, where the X in Sparse-X specifies how many user actions we generate in the initial action list before inserting a unique name. For each optimization, we compare against a baseline that only lacks the optimization in question; In particular, for mixed, we use the stable optimization for both lines; for stable, we use mixed memoization for both lines.

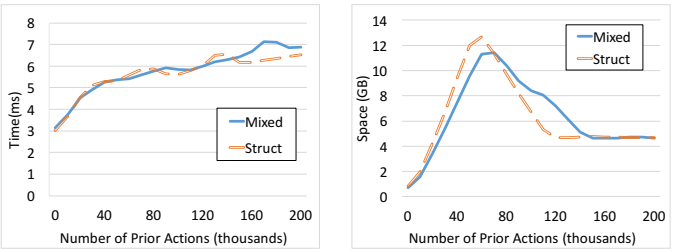
All experiments were compiled with rustc 1.9.0-nightly (e91f889ed 2016-03-03) release mode, and run on a 2.5 GHz MacBook Pro with 16 GB of RAM running Mac OS X 10.10.5. Each experiment is run for 200k actions. We present this data as a local average of the values before and after any plotted point. Unless noted otherwise, each plotted point represents an average of 20k actions.



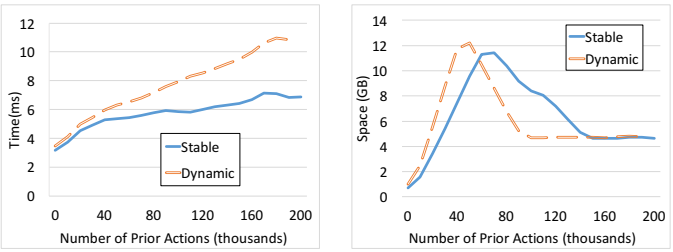
(a) Simple versus Responsive: Time (left) and Space (right)



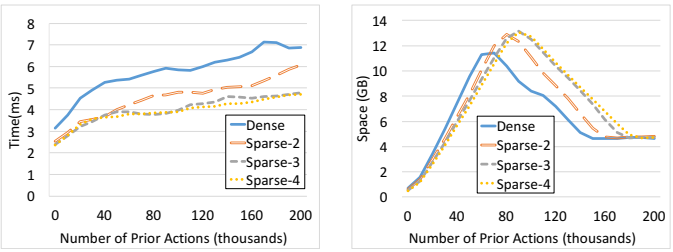
(b) Local edits versus global edits: Time (left) and Space (right)



(c) Mixed versus structural identity: Time (left) and Space (right)



(d) Stable versus dynamic nodes: Time (left) and Space (right)



(e) Varying sparseness setting: Time (left) and Space (right)

Figure 18: results

Simple versus responsive implementations. Figure 18a shows the time (left) and space (right) for running the simple and responsive implementations on the local and global workloads described above. In order to show crossover points, this plot only shows 20k actions, 10% of the plots discussed below. For the first 2k actions, the two global workload lines are each linear, due to the setup actions described above. At 3k actions, the Simp-Global line reaches 80ms, far outside the plotted bounds. By contrast, the Resp-Global line at 3k actions is 3.3ms, and it does not reach or exceed 5ms until 40k actions. Figure 17 shows that at 190k-200k actions, over 75% of the measured actions are under 10ms, with very few exceeding 15ms.

Initially, the Simp-Local and Resp-Local lines are lower than the Global lines, due to the lower total character count created by their action distribution. Initially, Simp-Local outperforms Resp-Local, but they cross over at only 2k actions. The lack of cached results yields a linear trend for the simple implementation, reaching 8ms after 16k actions.

The lack of cached data means our simple implementation requires very little memory: It uses no more than 40MB regardless of number of past interactions or workload. By contrast, the responsive version requires nearly 1GB for the local-edit workload, and more memory for the global-edit workload, since it caches the search path for each switch and goto action.

Local-edit versus global-edit workloads. Figure 18b shows time and space requirements for our responsive implementation of IC-Edit. The global workload takes only a few milliseconds more than the local workload, which is far less demanding. The initial trend in the global plot is sub-linear; in the right hand side, we observe a mild impact from extreme memory pressure.

The memory consumption of the global workload approaches the maximum for our test system (16GB). The operating system responds by paging Adapton's cache from RAM into swap space. This paging results in a peak memory level followed by a plateau around 5GB in this and all plots discussed below. That this paging only mildly impacts responsiveness suggests that IC-Edit only uses a small fraction of the cache provided by Adapton. We discuss this point and its implications in more detail below.

Adapton Optimizations. Figure 18c compares structural and mixed memoization. The plots show no significant time difference, but our mixed implementation uses less memory than the structural-only version. At 40k actions, for example, we measure space usage of 7.4GB and 9.4GB respectively.

Figure 18d compares the stable-node optimization to the dynamic-only version, where we observe significant time and space improvements. At 160k prior interactions, the dynamic-only version takes 50% more time; at 40k actions, it takes 50% more space.

Finally, Figure 18e compares the different sparsity settings. As the plot shows, sparseness improves both space and time. Though we experimented with higher sparseness settings, we did not observe further performance improvements. We conclude that sparseness is promising, but its use in IC-Edit requires more investigation to fully understand.

Current Limitations: Space Requirements. As discussed above, the simple implemention of IC-Edit consumes little memory compared to the responsive implementation, whose memory consumption is significant. Though not plotted above, we routinely observed that total memory (as estimated by Mac OS X), reached as high as 50GB at 200k actions. However, the space results in Figure 18 also suggest that future work can dramatically reduce the memory required by IC-Edit, and similar applications of the PRY methodology: While we observe that total memory (swap and resident) grows monotonically, after the system begins using swap space, resident memory flattens without adversely impacting the corresponding response times. This consistently-observed behavior suggests that future work can systemize aggressive, preemptive cache eviction within the Adapton library, rather than the OS, eliding the need for system-level swap space. Exploring this potential is outside the scope of the current paper, but instead remains as exciting future work.

7. Related Work

Reactive Computing. Reactive programming languages offer abstractions for processing events generated by dynamic environments. Early examples of event-based reactive environments for real-time systems in embedded software include Signal (Guernic et al. 1986) and Lustre (Caspi et al. 1987). Functional Reactive Programming (FRP) is a declarative programming model for constructing interactive applications (Elliott and Hudak 1997; Nilsson et al. 2002; Wan and Hudak 2000; Mandel and Pouzet 2005; Cooper and Krishnamurthi 2006; Ignatoff et al. 2006; Meyerovich et al. 2009; Courtney 2001; Krishnaswami 2013; Czaplicki and Chong 2013; Demetrescu et al. 2014; Boussinot and Susini 1998).

Arguably, the chief aim of FRP is to provide a declarative means of specifying programs whose values are time-dependent (stored in signals). In particular, each reactive language and FRP approach offers a set of building blocks for creating interactive systems with stateful feedback. Unlike PRY, these approaches do not define state as "the history of all past inputs", nor do they offer general-purpose IC, which seems necessary for a responsive implementation under such definitions of state. However, FRP abstractions do appear applicable for defining the "outer feedback loop" illustrated in Section 1, including a PRY-based system's interface to the underlying system's input and output signals.

Incremental computing. Researchers have provided various language-based approaches to incremental computation (Acar et al. 2006; Hammer and Acar 2008; Acar and Ley-Wild 2009; Hammer et al. 2015b). In particular, researchers have shown that for certain algorithms, inputs, and classes of input changes, IC delivers large, even *asymptotic* speed-ups over full reevaluation (Acar et al. 2007, 2008). IC has been developed in many different language settings (Shankar and Bodik 2007; Hammer et al. 2007, 2009; Chen et al. 2014b), and has addresses open problems in computational geometry (Acar et al. 2010).

Some PL approaches to IC are *static*, transforming programs to derive a second program that can process input changes. Static approaches perform these transformations a priori, before any dynamic changes. As such, static approaches are often lack the ability to transform general recursion or to fully cache and exploit dynamic dependencies (Liu and Teitelbaum 1995; Liu et al. 1998; Cai et al. 2014).

In contrast to static approaches, dynamic approaches attempt to trade space for time savings. A variety of dynamic approaches to IC have been proposed. Most early approaches fall into one of two camps: they either perform function caching of pure programs (Bellman 1957; McCarthy 1963; Michie 1968; Pugh 1988), or they support input mutation and employ some form of dynamic dependency graphs, along with a mechanism for performing *change propagation* (Acar et al. 2004, 2006; Hammer and Acar 2008; Acar and Ley-Wild 2009; Hammer et al. 2015a) Earlier work restricted programs to those expressible as *attribute grammars* (Demers et al. 1981; Reps 1982a,b; Vogt et al. 1991). Various threads of research propose general schemes for practical memoization, either making it applicable in more settings, or more efficient. Researchers have extended memoization to extend memoization to parallel C and C++ programs Bhatotia et al. (2015), to extend memoization to distributed, cloud-based settings Bhatotia et al. (2011), and have reduced the (often large) space overhead Chen et al. (2014a).

8. Conclusion

We propose the PRY methodology, which encourages developers to program interactive systems in two phases: The first phase advocates a purely-functional design whose state consists of all prior user inputs; the second phase seeks a responsive model by leveraging clever functional data structures, and techniques for general-purpose incremental computation (IC). We demonstrate PRY with a proof-of-concept text editor IC-Edit that we show is simple, specified by a formal semantics, and responsive (under 8ms, on average). To achieve this performance, we develop a novel functional data structure, the random access zipper (RAZ); it admits log-time random access, constant-time updates, and is amenable to IC. We also implement Adapton in Rust, allowing PRY-based programs to eschew garbage collection and experiment with a radical "cache everything" approach. Overall, we find strong evidence that future work can reduce swap space requirements with library-level (rather than system-level) cache eviction policies. Finally, we significantly lower the time and space requirements of Adapton by introducing several novel optimizations, and evaluate each empirically.

References

M. Abbott, T. Altenkirch, C. McBride, and N. Ghani. D for data: Differentiating data structures. *Fundam. Inf.*, 65(1-2):1–28, 2004.

U. A. Acar and R. Ley-Wild. Self-adjusting computation with Delta ML. In *Advanced Functional Programming*. 2009.

U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, 2003.

U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive memoization. Technical Report CMU-CS-03-208, Carnegie Mellon University, Nov. 2004.

U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. A library for self-adjusting computation. *ENTCS*, 148 (2), 2006.

U. A. Acar, A. Ihler, R. Mettu, and O. Sümer. Adaptive Bayesian inference. In *Neural Information Processing Systems (NIPS)*, 2007.

U. A. Acar, G. E. Blelloch, K. Tangwongsan, and D. Türkoğlu. Robust kinetic convex hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*, Sept. 2008.

U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Dynamic well-spaced point sets. In *Symposium on Computational Geometry*, 2010.

R. Bellman. *Dynamic Programming*. Princeton Univ. Press, 1957.

P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: MapReduce for incremental computations. In *SOCC*, 2011.

P. Bhatotia, P. Fonseca, U. A. Acar, B. B. Brandenburg, and R. Rodrigues. iThreads: A threading library for parallel incremental computation. In *ASPLOS*, 2015.

F. Boussinot and J.-F. Susini. The SugarCubes Tool Box: a Reactive Java Framework. *Software: Practice and Experience*, 28(14): 1531–1550, 1998.

Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *PLDI*, 2014.

P. Caspi, P. Pilaud, N. Halbwachs, and J. Plaice. Lustre, a Declarative Language for Programming Synchronous Systems. In *POPL*, pages 178–188, 1987.

Y. Chen, U. A. Acar, and K. Tangwongsan. Functional programming for dynamic and large data with self-adjusting computation. In *ICFP*, 2014a.

Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar. Implicit self-adjusting computation for purely functional programs. *J. Functional Programming*, 24(1):56–112, 2014b.

G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP*, 2006.

A. Courtney. Frappé: Functional Reactive Programming in Java. In *PADL*, pages 29–44, 2001.

E. Czaplicki and S. Chong. Asynchronous functional reactive programming for guis. In *PLDI*, volume 48, pages 411–422. ACM, 2013.

A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation of attribute grammars with application to syntax-directed editors. In *POPL*, 1981.

C. Demetrescu, I. Finocchi, and A. Ribichini. Reactive imperative programming with dataflow constraints. *ACM Trans. Program. Lang. Syst.*, 37(1):3:1–3:53, 2014.

C. Elliott and P. Hudak. Functional Reactive Animation. In *ICFP*, pages 263–273, 1997.

P. L. Guernic, A. Benveniste, P. Bournai, and T. Gautier. SIGNAL - A Data Flow-Oriented Language for Signal Processing. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34(2): 362–374, 1986.

M. Hammer and U. A. Acar. Memory management for self-adjusting computation. In *ISMM*, 2008.

M. Hammer, U. A. Acar, M. Rajagopalan, and A. Ghuloum. A proposal for parallel self-adjusting computation. In *DAMP '07: Declarative Aspects of Multicore Programming*, 2007.

M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: a C-based language for self-adjusting computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

M. A. Hammer, Y. P. Khoo, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. In *PLDI*, 2014.

M. A. Hammer, J. Dunfield, K. Headley, N. Labich, J. S. Foster, M. Hicks, and D. Van Horn. Incremental computation with names. 2015a.

M. A. Hammer, J. Dunfield, K. Headley, N. Labich, J. S. Foster, M. Hicks, and D. Van Horn. Incremental computation with names (extended version). arXiv:`1503.07792 [cs.PL]`, 2015b.

G. Huet. The zipper. *Journal of Functional Programming*, 1997.

D. Ignatoff, G. H. Cooper, and S. Krishnamurthi. Crossing State Lines: Adapting Object-Oriented Frameworks to Functional Reactive Languages. In *FLOPS*, pages 259–276, 2006.

N. R. Krishnaswami. Higher-order reactive programming without spacetime leaks. In *International Conference on Functional Programming (ICFP)*, Sept. 2013.

Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, 1995.

Y. A. Liu, S. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1998.

L. Mandel and M. Pouzet. ReactiveML, a Reactive Extension to ML. In *PPDP*, pages 82–93, 2005.

N. D. Matsakis and F. S. Klock II. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, 1963.

L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a Programming Language for Ajax Applications. In *OOPSLA*, pages 1–20, 2009.

D. Michie. “Memo” functions and machine learning. *Nature*, 218: 19–22, 1968.

H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, Oct. 2002. ACM Press.

W. Pugh. *Incremental Computation via Function Caching*. PhD thesis, Cornell University, 1988.

W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *POPL*, 1989.

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

T. Reps. *Generating Language-Based Environments*. PhD thesis, Cornell University, Aug. 1982a.

T. Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *POPL*, 1982b.

A. Shankar and R. Bodik. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *Programming Language Design and Implementation*, 2007.

H. Vogt, D. Swierstra, and M. Kuiper. Efficient incremental evaluation of higher order attribute grammars. In *PLILP*, 1991.

Z. Wan and P. Hudak. Functional Reactive Programming from First Principles. In *PLDI*, pages 242–252, 2000.