Tiny Structure Editors for Low, Low Prices!

(Generating GUIs from `toString` Functions)

Brian Hempel
University of Chicago
brianhempel@uchicago.edu

Ravi Chugh
University of Chicago
rchugh@uchicago.edu

***Abstract*—Writing `toString` functions to *display* custom data values is straightforward, but building custom interfaces to *manipulate* such values is more difficult. Though tolerable in many scenarios, this difficulty is acute in emerging value-centric IDEs—such as those that provide programming by examples (PBE) or bidirectional transformation (BX) modalities, in which users manipulate output values to specify programs.**

We present an approach that automatically generates custom GUIs from ordinary `toString` functions. By tracing the execution of the `toString` function on an input value, our technique overlays a *tiny structure editor* upon the output string: UI widgets for selecting, adding, removing, and modifying elements of the original value are displayed atop appropriate substrings.

We implement our technique—in a tool called TSE—for a simple functional language with custom algebraic data types (ADTs), and evaluate the tiny structure editors produced by TSE on a selection of existing and custom `toString` functions.

I. Introduction

Programmers often write `toString` functions to help interpret and debug code involving custom data types. For example, for a type of values describing numeric intervals, the string `"(-∞,10]"` conveys the meaning "all numbers less than or equal to 10" more succinctly than the string `"Interval(NegInf(), Before(10, True))"`, which might be a default serialization provided by the language.

Custom `toString` functions are usually straightforward to write, but what if the programmer needs not only to display the value but also *edit* the value as well? This need arises in emerging development environments such as bidirectional programming (discussed below) that rely on editing program values instead of code.

One idea is for the programming environment to enrich default string representations with automatically-generated, type-directed GUI widgets. For example, given the default representation `"Interval(NegInf(), Before(10, True))"`, the system might render a slider for "scrubbing" `10` to different values ([1], [2]) and a widget to select `NegInf()` and toggle it to `After(0, False)`.

Ideally, however, the domain-specific representation `"(-∞,10]"` would be editable, not just the default representation. Unfortunately, creating an *editable* domain-specific representation of values is considerably more difficult than writing `toString` functions for display.

978-1-7281-6901-9/20/$31.00 ©2020 IEEE

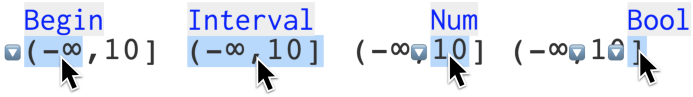

Fig. 1. Selection regions and UI elements generated by tracing the execution of the `toString` function for an interval data type.

Our Approach: Tiny Structure Editors (TSE)

We design a system, called TSE, that given a `toString` function for a custom data type, automatically generates *tiny structure editors* for manipulating values of that type.

To do so, TSE instruments the execution of the `toString` function applied to a value, and then overlays UI widgets on top of appropriate locations in the output string (Figure 1). To determine these locations, TSE employs two key technical ideas: (a) a modified string concatenation operation that preserves information about substring locations and (b) runtime dependency tracing (based on Transparent ML [3]) to relate those substrings to parts of the input value.

We implement TSE for a simple functional language with algebraic data types (ADTs), and we discuss the tiny structure editors that TSE produces.

Potential Applications

TSE is currently a prototype, proof-of-concept tool. However, we believe our approach would benefit a number of emerging techniques that allow programmers to specify code via direct manipulation of program values.

Literals in a Structure Editor. In *structure editors*—such as the Cornell Program Synthesizer [4]—and *block-based editors*—such as Scratch [5]—tree transformations rather than raw text edits are used to manipulate the program. Structure editors can use domain-specific representations for display. For example, the Barista [6] editor for Java shows rich, custom, type-specific views for mathematical and logical expressions in code. For editing, however, Barista falls back to ordinary textual manipulation. TSE could instead provide custom mini structure editors within the main structure editor.

Programming by Examples (PBE). Given input-output examples, these systems (e.g. [7], [8]) synthesize a small program. Sometimes many examples are required: Myth [8] requires 20 examples to synthesize binary tree insertion. Providing so many examples in text form can be cumbersome.

Direct-Manipulation Programming. Several tools augment text-based coding with direct manipulation of output values.

Bidirectional programming (BX) systems allow users to edit numbers ([9], [10], [11], [12]), strings ([13], [14], [11], [12]), or lists ([12]) in the output of a program to thereby change appropriate literals in the original code.

Compared to these BX systems, *output-directed programming (ODP)* systems allow the user to make larger, structural changes to the program ([15], [16], [2], [17], [18]), performing refactorings or inserting chunks of new code. To date, ODP systems carefully implement bespoke, domain-specific interfaces to enable selection and manipulation of the output.

Related Work

Each of the programming interactions above would benefit from an easy way to create domain-specific interfaces for custom data types. How do programmers currently input and edit program values in such systems?

Parse Functions. Programming is largely a text-based activity; entering values via text is thus a natural interface, but requires a parser. Custom parsers can be integrated with a language pre-processor like Template Haskell [19] or typed literal macros [20]. But the difficulty of writing a parser may not be worth the gain in expressiveness over the language's default value parser. TSE provides a structure editor on a domain-specific representation, without the labor of writing a parser.

Handcrafted GUIs. If interaction is important, the programmer may opt to manually craft a custom graphical user interface for their data type. Although this effort is justifiable for common types, e.g. colors or regular expressions [21], writing a custom UI may not be worth the trouble for one-off data types.

String Tracing. Some previous systems ([13], [14]) trace string operations, enabling developers to directly edit HTML output and thereby modify appropriate literal strings in the source PHP or Javascript. TSE also relies on tracing, but uses a more generic mechanism [3], allowing TSE to track how substrings relate to *any* value of interest, rather than just string literals.

II. Our Approach

Our approach, tiny structure editors (TSE), uses a custom program evaluator to instrument the execution of a programmer-provided `toString` function. TSE displays the string output and overlays UI widgets over appropriate substrings, allowing the user to modify the original value, but by interacting with the domain-specific representation generated by the `toString` function. Our TSE prototype supports `toString` functions over programmer-defined *algebraic data types (ADTs)* in an ordinary functional language similar to Elm (`https://elm-lang.org/`).

Algebraic Data Types (ADTs)

Somewhat analogous to inheritance in object-oriented languages, *algebraic data types (ADTs)* enumerate the variants of a type and the data associated with each variant [22]. Unlike an object, an ADT value is raw data, separate from the functions that operate on it. Because ADTs succinctly describe the variants of plain data, mainstream languages are adopting ADTs: "enums" in Swift and Rust are ADTs, as are "case classes" in Scala and "discriminated unions" in Typescript.

```
type Begin    = NegInf()          | After(Num, Bool)
type End      = Before(Num, Bool) | Inf()
type Interval = Interval(Begin, End)

toString : Begin -> String
toString(begin) = case begin of
  NegInf()             -> "(-∞"
  After(num, isClosed) ->
    (if isClosed then "[" else "(") ++ toString(num)

toString : End -> String
toString(end) = case end of
  Inf()                -> "∞)"
  Before(num, isClosed) ->
    toString(num) ++ (if isClosed then "]" else ")")

toString : Interval -> String
toString(interval) = case interval of
  Interval(begin, end) ->
    toString(begin) ++ "," ++ toString(end)

valueOfInterest : Interval
valueOfInterest =
  Interval(NegInf(), Before(10, True))
```

Fig. 2. ADT and `toString` definitions for a custom interval type.

Figure 2 shows three ADT definitions comprising a custom interval data type. The lower bound of an interval (`Begin`) has two variants representing whether the bound is negative infinity (`NegInf()`) or finite (`After(...)`). If finite, the bound records the finite boundary number and a boolean indicating whether the boundary is or is not included in the interval (is or is not *closed*). The type describing upper boundaries (`End`) is similar. An interval (`Interval`) is a lower and upper boundary together. The first word of each variant (`NegInf`, `After`, `Before`, `Inf`, `Interval`) is a *constructor* which acts as a function to create a *value* of the ADT. The last line of Figure 2 uses these constructors to create an interval value representing $(-\infty, 10]$. Data inside ADT values is extracted using "pattern matching" in *case splits* (i.e. switch statements) which define the handling of alternative variants, as shown in the `toString` functions in Figure 2.

Algorithm

Our automatic algorithm for generating tiny structure editors proceeds in three steps. The tracing evaluator relates substrings to portions of the original value, then 2D spatial regions over the rendered string are computed, and finally actions are assigned to the 2D regions.

1) Dependency Tracing: TSE utilizes a custom evaluator that traces dependency provenance, following Transparent ML (TML) [3]. The value of interest and its subvalues are first tagged with *projection paths* (e.g. 2.2.•) indicating their location within the value of interest:

$$\texttt{Interval(NegInf()}^{\{1.\bullet\}}\texttt{,Before(10}^{\{2.1.\bullet\}}\texttt{,True}^{\{2.2.\bullet\}}\texttt{)}^{\{2.\bullet\}}\texttt{)}^{\{\bullet\}}$$

Based on the value's type, the appropriate `toString` function is invoked on the value of interest and the tracing evaluator

propagates the dependency tags. Additionally, in TSE, string concatenation operations (++) do not produce a new, flattened string. Instead, the concatenation is deferred, resulting in a binary tree of substrings when evaluation completes (Figure 3). Because of the tracing evaluator, each substring and each concatenation carries a set of projection paths, relating parts of the string to parts of the value of interest (Figure 3b).

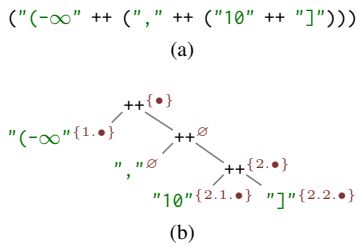

Fig. 3. A concatenation tree (two views).

2) Spatial Regions: In the final display, selection regions and UI widgets will be overlaid on top of the rendered string. To generate the selection regions, the string concatenation binary tree is translated into a binary tree of nested 2D polygons, with each polygon encompassing the spatial region of the associated substring (Figure 4a). Only regions associated with at least one path will ultimately be relevant (Figure 4b). Although nested, the regions are positioned flush without padding (Figure 4c), which can cause occlusion (discussed later). For a multiline string, the regions are shrunk to exclude whitespace, and each region may also exclude a portion of its first and last line (Figure 4d).

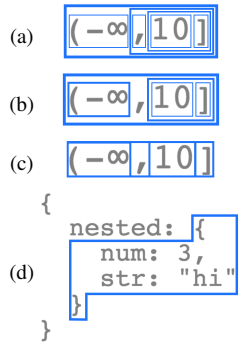

Fig. 4. String concatenations are translated into 2D regions atop the rendered string.

3) Selections and Actions: Once 2D regions of the displayed string are associated with corresponding locations in the value of interest, these 2D regions can be used to facilitate a number of interactions. Our TSE prototype explores three: (a) *selection* of subvalues; (b) *base value editing* of numbers and strings; and (c) *structural transformations*, namely item insertion, item removal, and constructor swapping.

Selection regions. When the user moves their cursor over the rendered string, the deepest (equivalently, smallest) region under their mouse is offered for selection/deselection. For the interval example, there are four possible selection regions, shown in Figure 1. Selection is currently inert, but the selection regions are the basis for positioning UI widgets. In the future, selection might facilitate cut-copy-paste operations, as in Vital [15], or might open a floating menu of possible code refactorings, as in Sketch-n-Sketch [18].

Editing base values. Literal numbers or strings from the value of interest may pass through to the output unchanged, for example the number `10` in the interval example. TSE lets the user manipulate these values. The user may click a number and drag up and down to scrub [1] the number to a different value. Both numbers and strings can be double-clicked to reveal a standard text box to text edit the value.

Structural transformations. Because an ADT definition describes the allowable structures for a value, TSE is able to infer possible transformations on the value of interest. For the interval example, the `Begin`, `End`, and `Bool` types each have an alternative constructor which can be toggled by clicking the change constructor button (▽) drawn to the left of the appropriate subvalue (Figure 1). These buttons allow the user to, e.g., change the lower bound from $-\infty$ to a finite bound (0 by default), or to toggle the boolean thus changing a finite boundary from closed (`"]"`) to open (`")"`). Which buttons to display are based on the selection region for the current mouse position—the deepest (smallest) region under the cursor. Since deepest regions may completely occlude some of their ancestors, TSE also displays the change constructor buttons for any such ancestor region that has no selectable area. For example, the `End` value `"10]"` is completely occluded by the `Num` `"10"` and the `Bool` `"]"`, so when the cursor is over the `Num` or `Bool` TSE shows the change constructor button for `End` (the ▽ over the comma in the right two cases in Figure 1).

For recursive ADTs such as lists or trees, TSE additionally draws buttons to insert (⊕) or remove (✕) items from the data structure, as shown in Figure 5 for a list. Remove buttons are associated with item(s) to be removed. Insert buttons are trickier to position—TSE must predict where an item not currently in the data structure will appear. This prediction is occasionally imprecise, as evaluated below.

Finally, if multiple buttons would be rendered on top of each other, these overlapping buttons are coalesced into a single button that will open a menu with the different transformations.

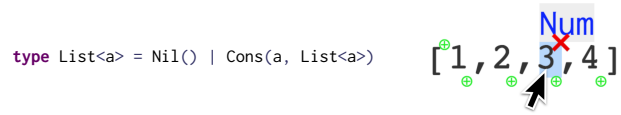

Fig. 5. Generated GUI for list `Cons(1,Cons(2,Cons(3,Cons(4,Nil()))))`.

Additional Tracing Details

To increase the chance that an unmodified `toString` function will generate a sensible structure editor, we made minor changes to TML [3], which we mention for completeness.

In ordinary TML, certain constant substrings, such as the opening `"["` and closing `"]"` of a list, are not dependent on the list because they are always shown. To associate these constant delimiters with the appropriate value, TSE tags the entire result of any `toString` call as dependent on its argument. For similar scenarios that do not occur at `toString` boundaries, TSE also offers a `basedOn(dep,x)` primitive that the `toString` author may use to add `dep` to the dependencies of `x`. Only one of our case studies below utilized this `basedOn` primitive.

On the other hand, to avoid extraneous dependencies, prefix and suffix strings shared by *all* branches of a case split are pulled outside the case split—otherwise, these constant substrings would be marked as dependent on the the value being split on. This normalization happens transparently before every execution and is not displayed to the user. Our supplementary technical report [25] includes an example of this scenario.

Data Structure	Description	%Selectable Subvalues	%Selectable Items	%Reasonable Inserts	Notes
Interval	`"(-∞,10]"`	80% (4/5)			
Date	`"May 9, 2020"`	100% (4/4)			Components represented separately.
JSON (multiline)	w/arrays, objects, strings, nums	33% (14/43)		81% (13/16)	`basedOn` used 3x.
List	`"[1,2,3]"`	86% (6/7)	100% (3/3)	100% (4/4)	
List (`"]"` in base case)	`"[1,2,3]"`	100% (7/7)	100% (3/3)	100% (4/4)	
List (via `join`)	`"[1,2,3]"`	71% (5/7)	100% (3/3)	100% (4/4)	
List (via different `join`)	`"[1,2,3]"`	86% (6/7)	100% (3/3)	100% (4/4)	
Tree (S-exp)	`"(2 (1) (4 (3) (5)))"`	53% (10/19)	100% (5/5)	14% (2/14)	5 inserts missing; poor placements.
Tree (indented hierarchy)	`"2\n  1\n  4\n    3\n    5"`	21% (4/19)	100% (5/5)	21% (3/14)	5 inserts missing; shared placements.
Pair [23]	`"(10,"ten")"`	100% (3/3)			
List [23]	`"[1,2,3]"`	100% (7/7)	100% (3/3)	100% (4/4)	
ADT (recursive) [23]	`"Ctor4 (Ctor3 True "asdf")"`	100% (4/4)		50% (1/2)	`Bool` region too long; same insert 2x.
Record [23]	`"Record {field1 = ..., ...}"`	100% (9/9)			`Bool` region too long.
Set [24]	`"fromList [2,3,5,7]"`		100% (4/4)	0%	Not 1-to-1 w/ADT definition.

Fig. 6. Case studies of hand-written and translated `toString` functions.

III. Case Studies

TSE's goal is to provide low- to no-cost domain-specific value editors. We tested TSE on `toString` functions for a number of datatypes, measuring several properties of the generated editors as shown in Figure 6. Figure 6 reports the percentage of ADT subvalues that could be directly selected (i.e. were not occluded, missing, or sharing a selection region with other subvalues). For data types representing containers (e.g. lists or sets), Figure 6 reports the percentage of contained items that can be selected. To evaluate TSE's heuristic for insert button positioning, Figure 6 also reports the percentage of insert transformations placed in reasonable locations. An insert transformation is considered *un*reasonably placed if either (a) the insert should be possible but is not assigned to any button in the UI, or (b) the insert shares a single button with other inserts, or (c) clicking the button inserts an item at a location other than the button's position.

To provide evidence that TSE can operate on unmodified `toString` functions, we translated several `toString` functions from Haskell's standard libraries to our Elm-like language, as shown in the bottom half of Figure 6. These translations were performed as literally as possible.

The case studies revealed a few issues to address in subsequent versions of TSE. Most notably, zero-width regions such as those from empty strings are ignored, which for some variants of list `toString` caused the final `Nil()` to be un-selectable. Additionally, selection region sharing and occlusion are sometimes troublesome. Two subvalues sharing the same selection region is a less of an issue—depending on the application, selecting a shared region could offer to operate on any of the associated items. Occlusion, however, results in certain subvalues being un-selectable. One solution might be to expand ancestor regions by a few pixels, resulting in regions more like Figure 4b rather than Figure 4c. Finally, insert buttons could be better placed for tree-like data structures, but, as discussed next, how best to handle actions is a domain-specific consideration.

IV. Discussion

TSE generates structure editors based on the `toString` function for a value, with little to no further programmer effort required. We envision value-centric programming systems that offer editable, domain-specific representations for custom data types, thus affording the programmer a more natural interface for specifying changes on the operation of their program.

At present, we implemented our TSE prototype independent of any of these possible settings. Applying TSE to a particular application will require a number of further design decisions, particularly surrounding the handling of actions. For example, consider the set data structure in Figure 6. The reference implementation [24] is based on a tree and maintains a number of invariants such as balancing, ordering, and non-duplication. None of these invariants are expressible in a standard ADT definition alone, and the internal tree structure is not exposed in the `toString` output (`"fromList [2,3,5,7]"`). Therefore, only some of TSE's selection regions are relevant—namely, the terminal items, as reported in Figure 6—and the structural transformations generated by TSE are not meaningful because they do not enforce the set invariants. TSE does not yet provide an interface for specifying custom insert and remove functions, instead such an interface might be part of a larger, future IDE.

Beyond action handling for data with complex invariants, our prototype has a number of minor limitations. First, systems that rely on string tracing ([13], [14]) provide custom implementations of string manipulation functions that correctly propagate dependencies. We currently only support string concatenation and string length—supplementing our language with additional string functions remains future work. Finally, our core language and TML do not support nested pattern matches. How dependency semantics should work for nested patterns is an open question—although a language's compiler will unnest the patterns [26], different unnestings can result in different dependency traces. While not uncommon, such ambiguous cases did not occur in our examples.

Adapting TSE to the more common object-oriented setting will require different tracing semantics, because "variants" are handled by virtual method lookups rather than case splits.

Further details about TSE's algorithm and heuristics are available in a supplementary technical report [25].

Acknowledgments

Our thanks to Andrew McNutt and the reviewers, whose suggestions helped improve this paper. This work was supported by U.S. National Science Foundation Grant No. 1651794 (*Direct Manipulation Programming Systems*).

References

[1] Victor, Bret, "Scrubbing Calculator," 2011, http://worrydream.com/ScrubbingCalculator/.

[2] McDirmid, Sean, "A Live Programming Experience," in *Future Programming Workshop, Strange Loop*, 2015, https://onedrive.live.com/download?cid=51C4267D41507773&resid=51C4267D41507773%2111492&authkey=AMwcxdryTyPiuW8 https://www.youtube.com/watch?v=YLrdhFEAiqo.

[3] U. A. Acar, A. Ahmed, J. Cheney, and R. Perera, "A Core Calculus for Provenance," *Journal of Computer Security*, 2013.

[4] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM (CACM)*, 1981.

[5] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: Programming for All," *Communications of the ACM (CACM)*, 2009.

[6] A. J. Ko and B. A. Myers, "Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors," in *Human Factors in Computing Systems (CHI)*, 2006.

[7] S. Gulwani, "Automating String Processing in Spreadsheets Using Input-Output Examples," in *Symposium on Principles of Programming Languages (POPL)*, 2011.

[8] P.-M. Osera and S. Zdancewic, "Type-and-Example-Directed Program Synthesis," in *Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[9] K. Fukahori, D. Sakamoto, J. Kato, and T. Igarashi, "CapStudio: An Interactive Screencast for Visual Application Development," in *Conference on Human Factors in Computing Systems (CHI), Extended Abstracts*, 2014.

[10] R. Chugh, B. Hempel, M. Spradlin, and J. Albers, "Programmatic and Direct Manipulation, Together at Last," in *Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[11] K. Kwok and G. Webster, "Carbide Alpha," 2016, https://alpha.trycarbide.com/.

[12] M. Mayer, V. Kunčak, and R. Chugh, "Bidirectional Evaluation with Direct Manipulation," *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*, 2018.

[13] X. Wang, L. Zhang, T. Xie, Y. Xiong, and H. Mei, "Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis," in *International Symposium on the Foundations of Software Engineering (FSE)*, 2012.

[14] C. Schuster and C. Flanagan, "Live Programming by Example: Using Direct Manipulation for Live Program Synthesis," in *LIVE Workshop*, 2016.

[15] K. Hanna, "A Document-Centered Environment for Haskell," in *International Workshop on Implementation and Application of Functional Languages (IFL)*, 2005.

[16] C. D. Hundhausen and J. L. Brown, "What You See Is What You Code: A live Algorithm Development and Visualization Environment for Novice Learners," *Journal of Visual Languages and Computing*, 2007.

[17] R. Schreiber, R. Krahn, D. H. H. Ingalls, and R. Hirschfeld, *Transmorphic: Mapping Direct Manipulation to Source Code Transformations*, 2017.

[18] B. Hempel, J. Lubin, and R. Chugh, "Sketch-n-Sketch: Output-Directed Programming for SVG," in *Symposium on User Interface Software and Technology (UIST)*, 2019.

[19] T. Sheard and S. L. Peyton Jones, "Template Meta-programming for Haskell," *SIGPLAN Notices*, 2002.

[20] C. Omar and J. Aldrich, "Reasonably Programmable Literal Notation," *PACMPL*, no. ICFP, 2018.

[21] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers, "Active Code Completion," in *International Conference on Software Engineering (ICSE)*, 2012.

[22] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002, pp. 132–142.

[23] GHC Team, "Glasgow Haskell Compiler 8.6.5," 2019, https://gitlab.haskell.org/ghc/ghc/tree/ghc-8.6.5-release.

[24] Daan Leijen, "Data.Set," in *Haskell Containers Library*, 2002, https://hackage.haskell.org/package/containers-0.6.2.1/docs/src/Data.Set.html.

[25] B. Hempel and R. Chugh, "Tiny Structure Editors for Low, Low Prices! Technical Supplement," 2020.

[26] M. Baudinet and D. MacQueen, "Tree Pattern Matching for ML," 1985.

Tiny Structure Editors for Low, Low Prices! Technical Supplement

Brian Hempel
University of Chicago
brianhempel@uchicago.edu

Ravi Chugh
University of Chicago
rchugh@uchicago.edu

Abstract*—This supplement further details the techniques introduced in our VL/HCC 2020 paper *Tiny Structure Editors for Low, Low Prices! (Generating GUIs from* `toString` *Functions) [1].

This technical supplement describes further considerations relating to the main paper, Tiny Structure Editors for Low, Low Prices! (Generating GUIs from `toString` Functions) [1]. These further considerations are:

- Why did we build TSE (Section I)?
- A few algorithm details: how TSE normalizes string concatenation trees and how insert/remove buttons are positioned (Section II).
- An example of why TSE pulls shared prefixes/suffixes out of branches to produce better dependency structures (Section III).
- A brief discussion of how TSE might be applied to an object-oriented setting (Section IV).
- Formal evaluation semantics detailing how TSE adapts Transparent ML [2] (Section V).
- A discussion of dependency structure ambiguity in the presence of nested patterns (Section VI).

I. Why TSE?

TSE was born as the authors considered the future of output-directed programming (ODP), a paradigm in which the user directly manipulates program output to thereby generate ordinary code—code that can be text-edited at any time without loss of direct manipulation features. While ODP has been explored for limited manipulations on HTML output (e.g. [3], [4]) and more full-featured manipulations for graphical output (e.g. [5], [6]), ODP for *general purpose* program construction has been largely unexplored, apart from initial forays by Vital [7] and ALVIS Live [8]. If ODP is to be useful at all stages of general program construction, then all intermediate data structures during program execution must be visible and *manipulable*. How can an ODP system offer manipulable values when a default representation would be unwieldy but asking the programmer to learn a new GUI framework is unreasonable? Thus TSE was born.

II. Further Algorithm Details

A. Concatenation Tree Normalization

After `toString` execution produces a tree of substring concatenations (Figure 3 in the main paper), the projection path tags undergo normalization. Identical projection paths shared by adjacent substrings are recursively redistributed to their parent concatenation; afterwards nested occurrences of the same path are removed, retaining only outermost occurrences of a path. This normalization produces no change in the `Interval` example.

B. Positioning Remove and Insert Buttons

For recursive ADTs such as lists or trees, TSE draws buttons to insert or remove items from the data structure (as shown in Figure 5 in the main paper). Remove buttons (✕) are associated with the "contained" items that would disappear if a subvalue were replaced with one of its recursive children. For example, consider removing the item `2` from the list `[1,2,3]`, which desugars to:

`Cons(1,`**`Cons(2,`**<u>**`Cons(3,Nil())`**</u>**`)`**`)`

Removing the `2` means replacing the **bolded** subvalue above with its recursive child (<u>underlined</u>). Although the bolded **`Cons(...)`** is what is being replaced, that **`Cons(...)`** subvalue is itself a list and it would be inappropriate to imply that the whole sublist is what would be removed. Instead, the remove button is associated with that subvalue's non-recursive children, namely the **`2`**, that will disappear upon removal.

Insertion is roughly the reverse, and is similarly accomplished by looking for recursion in the ADT definition and using such locations as insertion points. Although generating candidate insertions is straightforward, positioning the insert buttons (⊕) is tricky because it relies on predicting where an item not currently in the data structure will appear. For this prediction, TSE uses the bottommost, rightmost point out of up to three candidate points: (1) the bottommost, rightmost point of the region(s) associated with the projection path immediately before the insertion location, (2) the topmost, leftmost point of the region(s) associated with the insert location projection path, and (3) the topmost, leftmost point of the region(s) associated with the projection path immediately after the insert location. Because a large, complicated data structure may include multiple kinds of containers of various types, these candidate points are subject to the additional constraint that they must be associated with the same container that is being inserted into. To enforce this constraint, container root paths are estimated by searching for projection paths whose parent value has a different type; all candidate points

must be prefixed by the same container root. As shown in Figure 5 in the main paper, TSE's positioning heuristic does not always place insert buttons in an aesthetically consistent location—the first insertion button is above the list while the others are below—but TSE's heuristic needs to handle empty and multi-line data structures and we found the above heuristic, relative to other heuristics we tried, was least likely to make confusing placements.

III. Code Normalization

To avoid extraneous dependencies during tracing execution, prefix and suffix strings shared by all branches of a case split are pulled outside the case split.

Consider the following version of a list `toString` function:

```
toString(list) = "[" ++ elemsToString(list) ++ "]"

elemsToString(list) = case list of
    Nil()            -> ""
    Cons(head, tail) -> case tail of
        Nil()     -> toString(head)
        Cons(_,_) -> toString(head) ++ "," ++ elemsToString(tail)
```

In the final case split, the split on `tail`, both branches call `toString(head)`. That head element will be marked as dependent on `tail`, and therefore, in the UI, moving the mouse over the head element will erroneously be interpreted as also referring to the tail—but the tail is the rest of the list *after* that element. Therefore, the code is automatically and transparently translated to...

```
elemsToString(list) = case list of
    Nil()            -> ""
    Cons(head, tail) -> toString(head) ++ case tail of
        Nil()     -> ""
        Cons(_,_) -> "," ++ elemsToString(tail)
```

...which removes the extraneous dependency, so that the head is not associated with the tail. This normalization happens transparently before every execution and is not displayed to the user.

IV. TSE for Object-Oriented Programming

How might TSE be applied to object-oriented (OO) programming? Instead of ADTs, variants in an OO setting can be encoded as subclasses of a shared abstract superclass. With ADTs, differences between variants are handled via case splits; with objects, each subclass defines its own version of a particular named method. Dynamic (i.e. virtual) method dispatch chooses the appropriate implementation for a particular object, subsuming the role of case splits in the ADT setting. The key tracing rule in TSE, inherited from TML, dictates that the result of a case split is marked as dependent on the value split upon (see EvalCase in Figure 8). In the interval example, this rule is responsible for 3 of the 5 dependency tags in the result and this rule is what allows TSE to offer meaningful change constructor actions. In an OO setting, this rule would be equivalent to marking the result of a dynamic call (i.e. a method defined in the subclasses) as dependent on the receiving object. The other tracing rules are equivalent in an OO setting. Finally, in an OO setting, actions for transforming objects are not so straightforward because objects encapsulate (hide) their implementation, whereas ADTs are plain data. As with the set example discussed in the paper, the broader programming system will need to provide a configuration interface for mapping object methods to semantic insert/remove/change actions in the UI.

V. A Dependency Provenance Algorithm

For tracing, TSE adapts the dependency provenance scheme of Transparent ML (TML) [2]. Provenance tracking in TML is ordinarily performed in two steps: first an execution trace is recorded during execution, then the desired provenance information is extracted from the trace. This two-step process enables TML to support multiple definitions of provenance. But because TSE only uses TML's *dependency* provenance scheme, we can simplify this process: we collapse the two steps together and record the dependencies directly during execution, thus foregoing the need to define a separate syntax of traces. The final provenance tags are still the same as in original TML (modulo the TSE-specific syntactic forms in Figure 7).

Figure 7 describes the syntax of TSE's core language. The traditional constructs in the expression language are: variables x, recursive functions $f(x).e$ (where f is the function name), functional application $e_1(e_2)$, constructors with multiple arguments $C(e_1, \ldots, e_n)$, case splits with multiple branches $\texttt{case}\ e\ \texttt{of}\ \overline{C_i(x_1, \ldots, x_n) \to e_i}$, strings s, numbers n, and numeric binary operations $e_1 \oplus e_2$. Surface language if-then-else statements are desugared to case splits on `True` and `False`. To track provenance for substrings, TSE's expression language includes several nonstandard primitive operations: string concatenation $e_1 +\!\!+ e_2$, string length inspection $\texttt{strLen}(e)$, number to string conversion $\texttt{numToStr}(e)$. TSE also supports manual dependency addition via $\texttt{basedOn}(e_d, e)$, which marks the result of e as dependent on that of e_d.

To track how output values are dependent on the input value of interest, Transparent ML (TML) assigns identifiers to each subvalue of the value of interest. These identifiers take the form of *projection paths* π, which denote the tree-descent path from the root of the value of interest to the identified subvalue. Each *(tagged) value* w carries a set of these paths indicating the subvalues of the value of interest that w depends on.

Expressions e	$::=$	$x \mid f(x).e \mid e_1(e_2)$
	$\mid$	$C(e_1, \ldots, e_n)$
	$\mid$	$\texttt{case}\ e\ \texttt{of}\ \overline{C_i(x_1, \ldots, x_n) \to e_i}$
	$\mid$	$s \mid e_1 +\!\!+ e_2 \mid \texttt{strLen}(e)$
	$\mid$	$n \mid e_1 \oplus e_2 \mid \texttt{numToStr}(e)$
	$\mid$	$\texttt{basedOn}(e_d, e)$
Projection Paths π	$::=$	$\bullet \mid i.\pi$
(Tagged) Values w	$::=$	$v^{\{\pi_1, \ldots, \pi_n\}}$
(Untagged) Pre-Values v	$::=$	$[E]\, f(x).e \mid C(w_1, \ldots, w_n)$
	$\mid$	$s \mid w_1 +\!\!+ w_2 \mid n$
	$\mid$	$\texttt{dyncall}(f)$
Tagged Environments E	$::=$	$- \mid E, x \mapsto w$

Fig. 7. Expressions, values, and, for dependency tracking, projection paths.

Evaluation with Dependency Provenance $\boxed{E \vdash e \Downarrow w}$

$$\text{[EvalVar]}\quad \frac{}{E \vdash x \Downarrow E(x)}$$

$$\text{[EvalDynVar]}\quad \frac{x \in \{\text{“}toString\text{''}, \text{“}showsPrecFlip\text{''}\}}{E \vdash x \Downarrow \texttt{dyncall}(x)^{\{\}}}$$

$$\text{[EvalFun]}\quad \frac{}{E \vdash f(x).e \Downarrow ([E]\, f(x).e)^{\{\}}}$$

$$\text{[EvalApp]}\quad \frac{E \vdash e_1 \Downarrow ([E_f]\, f(x).e)^{p_1} \qquad E \vdash e_2 \Downarrow w_2 \qquad E_f,\, f \mapsto ([E_f]\, f(x).e)^{p_1},\, x \mapsto w_2 \vdash e_f \Downarrow v^{p}}{E \vdash e_1(e_2) \Downarrow v^{p_1 \cup p}}$$

$$\text{[EvalDynApp]}\quad \frac{E \vdash e_1 \Downarrow \texttt{dyncall}(f)^{p_1} \qquad E \vdash e_2 \Downarrow {v_2}^{p_2} \qquad typeDispatch(f, v_2) = g \qquad E,\, x \mapsto {v_2}^{p_2} \vdash g(x) \Downarrow v^{p}}{E \vdash e_1(e_2) \Downarrow v^{p_1 \cup p\, (\cup\, p_2 \text{ if } f = \text{“}toString\text{''})}}$$

$$\text{[EvalCtor]}\quad \frac{\overline{E \vdash e_i \Downarrow w_i}}{E \vdash C(e_1\,, \ldots, e_n) \Downarrow C(w_1\,, \ldots, w_n)^{\{\}}}$$

$$\text{[EvalCase]}\quad \frac{E \vdash e \Downarrow C_j(w_1\,, \ldots, w_n)^{p} \qquad E,\, x_1 \mapsto w_1,\, \ldots,\, x_n \mapsto w_n \vdash e_j \Downarrow {v_j}^{p_j}}{E \vdash \texttt{case}\ e\ \texttt{of}\ \overline{C_i(x_1\,, \ldots, x_n) \rightarrow e_i} \Downarrow {v_j}^{p \cup p_j}}$$

$$\text{[EvalStr]}\quad \frac{}{E \vdash s \Downarrow s^{\{\}}}$$

$$\text{[EvalConcat]}\quad \frac{E \vdash e_1 \Downarrow w_1 \qquad E \vdash e_2 \Downarrow w_2}{E \vdash e_1 \mathbin{++} e_2 \Downarrow (w_1 \mathbin{++} w_2)^{\{\}}}$$

$$\text{[EvalStrLen]}\quad \frac{E \vdash e \Downarrow w \qquad strLen(w) = n \qquad allDepsDeep(w) = p}{E \vdash \texttt{strLen}(e) \Downarrow n^{p}}$$

$$\text{[EvalNum]}\quad \frac{}{E \vdash n \Downarrow n^{\{\}}}$$

$$\text{[EvalBinop]}\quad \frac{E \vdash e_1 \Downarrow {n_1}^{p_1} \qquad E \vdash e_2 \Downarrow {n_2}^{p_2} \qquad n_1 \oplus n_2 = v}{E \vdash e_1 \oplus e_2 \Downarrow v^{p_1 \cup p_2}}$$

$$\text{[EvalNumToStr]}\quad \frac{E \vdash e \Downarrow n^{p} \qquad numToStr(n) = s}{E \vdash \texttt{numToStr}(e) \Downarrow s^{p}}$$

$$\text{[EvalBasedOn]}\quad \frac{E \vdash e_d \Downarrow {v_d}^{p_d} \qquad E \vdash e \Downarrow v^{p}}{E \vdash \texttt{basedOn}(e_d\,, e) \Downarrow v^{p_d \cup p}}$$

Fig. 8. TSE's adaptation of TML semantics.

Initially, the value of interest and its subvalues are tagged with singleton sets identifying their locations[1], as in the example in the main paper:

$$\texttt{Interval(NegInf()}^{\{1.\bullet\}}\texttt{,Before(10}^{\{2.1.\bullet\}}\texttt{,True}^{\{2.2.\bullet\}}\texttt{)}^{\{2.\bullet\}}\texttt{)}^{\{\bullet\}}$$

These projection paths are propagated during evaluation.

Primitive values in TSE, called *(untagged) pre-values* v to distinguish them from their tagged forms w, include several traditional forms: recursive function closures $[E]\, f(x).e$ (where E is the captured environment of variable bindings and f is the function name), constructed ADT values $C(w_1\,, \ldots, w_n)$, simple strings s, and numbers n. The TSE-specific forms are deferred string concatenation $w_1 \mathbin{++} w_2$ (where w_1 and w_2 are each either deferred concatenations or simple strings), and a dynamic function call $\texttt{dyncall}(f)$ for late-bound type-based function dispatch to support multiple implementations of `toString` (discussed below).

Finally, for evaluation and closure capture, *tagged environments* E store a mapping from variable names to tagged values.

Figure 8 describes TSE's adaptation of the tracing evaluation semantics of Transparent ML (TML) [2]. The tracing evaluation relation $E \vdash e \Downarrow w$ takes a tagged environment E and expression e for input and produces a tagged value w as output.

Variable names are simply looked up in the execution environment (EvalVar). Values in the environment are already tagged with dependencies—these tags are retained unchanged. If the variable name is one of several special names that require type-based dynamic dispatch, instead of looking up the name in the environment, a $\texttt{dyncall}(x)$ value is produced representing the deferred function call (EvalDynVar). This value is assigned an empty set of dependencies. The variable name will be resolved to a function once the type of the argument is known (EvalDynApp).

Function definitions resolve to closures that capture the execution environment (EvalFun). The closure value has no dependencies. Function application (EvalApp) is standard. Functions are singly recursive[2]: after the argument expression

[1]Figure 11 in Acar et al. [2] formalizes this initial tagging operation, although in their setting the operation is a little less trivial: their projection paths allow lookups into variables in the environment because they define program input to be a full execution environment of variable bindings which may include function closures with nested environments—for `toString` tracing in TSE we assume the input is a single value without closures and thus our projection paths do not need to support variable lookups.

[2]Before execution, mutual recursion is desugared to single recursion, following Exercise 9 of https://caml.inria.fr/pub/docs/u3-ocaml/ocaml-ml.html#Exo-9

is evaluated, both the argument value and the function closure are added as new bindings into the captured environment and the function body is executed. After a function result is produced, the dependencies p_1 of the closure are unioned with the dependencies p of the function result (although typically p_1 is the empty set). Notably, the dependencies of the argument value w_2 are not included in the union—if the argument was used in the computation of the result, these dependencies will already be represented in p.

Type-based dynamic function dispatch (EVALDYNAPP) is dynamically resolved to ordinary function application. Dynamic dispatch allows the same function variable name to be defined multiple times, with different type annotations on each definition. The appropriate implementation is chosen when the function is called, based on the type of the argument. The implementation operates as follows: for those variable names considered dynamic[3], a preprocessing step on the code (not shown) renames those (colliding) variable definition names to unique names. An internal dictionary (not shown) remembers the association between the type annotation on each definition and its unique name. At the call site, when the argument value v_2 is produced, its type is inspected and the dictionary for the non-unique function name f is consulted ($typeDispatch$ in EVALDYNAPP) producing the unique name g of the implementation whose argument type matches. This function g is then applied normally to the argument. This scheme for dynamic dispatch means that multi-argument functions can only be dynamic in their first argument (although a fancy desugaring scheme might work around this limitation without changing the core semantics presented here). The dependencies for dynamic dispatch are propagated as in ordinary function application: the paths p_1 on the deferred function call `dyncall`(f) are merged with those paths p from the function result. For reasons discussed in the main paper—namely, to associate constant delimiters with the appropriate value—if the dynamic call was a `toString` function, then the dependencies p_2 of the argument are also added to those of the result value.

Constructor introduction (EVALCTOR) is standard—each argument expression is evaluated (the overline denotes multiplicity) and used for the arguments of the constructed value. The constructed value is assigned no dependencies.

Case splits (EVALCASE) are also standard. The scrutinee e is evaluated to a constructed value and the appropriate branch j is taken based on the scrutinee value's constructor. The constructor's arguments are bound to appropriate variable names for the branch and the branch expression e_j is evaluated. Finally, the dependencies p of the scrutinee value are unioned with the dependencies p_j of the branch result. This merger is key! Marking the case result as dependent on the scrutinee result is vital for TSE to work: this rule allows the UI to offer change constructor actions e.g. clicking to toggle a boolean.

String literals are assigned no dependencies (EVALSTR). Deferred string concatenation (EVALCONCAT) is analogous to constructor introduction (EVALCTOR)—the string concatenation operator $+\!\!+$ is essentially an infix constructor with two arguments. The concatenation is assigned no dependencies but the dependencies on the left and right children are preserved. Inspecting the string length (EVALSTRLEN) is a built-in operation whose resulting number is marked as dependent on *all* the dependencies throughout the whole string concatenation tree (gathered by $allDepsDeep$ in the rule).

Numeric operations are standard. Numeric literals (EVALNUM) are assigned no dependencies. Binary operations on numbers (EVALBINOP) produce a resulting value that is marked as dependent on both operands. Converting a number to a string (EVALNUMTOSTR) transfers dependencies from the number to the resulting string.

Finally, TSE supports manual dependency addition (EVALBASEDON) through the `basedOn`(e_d, e) built-in, which returns the result of its second argument e after adding the paths from the result of the first argument e_d. Manual dependency addition is occasionally useful for the same reason that `toString` results are marked as dependent on the `toString` argument: constant delimiters, being constant, are not normally associated with the item being delimited. The dependency can be manually added.

VI. Dependence Ambiguity Under Nested Patterns

TML [2] does not support nested patterns. How dependency should be defined for nested patterns is an open question. Consider the following nested pattern of tuples:

```
case boolPair of
    (True, True) -> a
    (_, _)       -> b
```

The pattern match compiler has two options for un-nesting the patterns. Either the first or the second element of the pair may be inspected before the other. Both are semantically equivalent:

```
case boolPair of
    (fst, snd) -> case fst of
        False -> b
        True  -> case snd of
            False -> b
            True  -> a


case boolPair of
    (fst, snd) -> case snd of
        False -> b
        True  -> case fst of
            False -> b
            True  -> a
```

While both versions are semantically equivalent, they can result in different dependency structures. If, e.g., `boolPair` is `(True, False)`, in the first version all case splits are executed and the result will be marked as dependent on all three scrutinees: `boolPair`, `fst` and `snd`. In the second version, the deepest case split is not executed and the result is thereby not marked as dependent on `fst`, only `boolPair` and `snd`.

This example suggests that relying on the pattern match compiler to determine the dependency structure may not be the right approach. An explicit dependency semantics for

[3]Currently the dynamic names are hard-coded in our prototype. There are two dynamic names: `toString` and, for the GHC examples, `showsPrecFlip`.

nested patterns might be required. Programmers think about case branches from top to bottom—if the first pattern does not match, try the second, and so on. Dependency semantics could match that intuition: a branch result might be marked as dependent on values corresponding to all nested patterns in the current *and prior* branches (of the same constructor).

References

[1] B. Hempel and R. Chugh, "Tiny Structure Editors for Low, Low Prices! (Generating GUIs from `toString` Functions)," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2020.

[2] U. A. Acar, A. Ahmed, J. Cheney, and R. Perera, "A Core Calculus for Provenance," *Journal of Computer Security*, 2013.

[3] X. Wang, L. Zhang, T. Xie, Y. Xiong, and H. Mei, "Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis," in *International Symposium on the Foundations of Software Engineering (FSE)*, 2012.

[4] M. Mayer, V. Kunčak, and R. Chugh, "Bidirectional Evaluation with Direct Manipulation," *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*, 2018.

[5] McDirmid, Sean, "A Live Programming Experience," in *Future Programming Workshop, Strange Loop*, 2015, https://onedrive.live.com/download?cid=51C4267D41507773&resid=51C4267D41507773%2111492&authkey=AMwcxdryTyPiuW8 https://www.youtube.com/watch?v=YLrdhFEAiqo.

[6] B. Hempel, J. Lubin, and R. Chugh, "Sketch-n-Sketch: Output-Directed Programming for SVG," in *Symposium on User Interface Software and Technology (UIST)*, 2019.

[7] K. Hanna, "A Document-Centered Environment for Haskell," in *International Workshop on Implementation and Application of Functional Languages (IFL)*, 2005.

[8] C. D. Hundhausen and J. L. Brown, "What You See Is What You Code: A live Algorithm Development and Visualization Environment for Novice Learners," *Journal of Visual Languages and Computing*, 2007.