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?

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.

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.

©2020 IEEE. This is the accepted version of the paper. The trivially revised final version is in VL/HCC 2020: https://ieeexplore.ieee.org/document/9127256

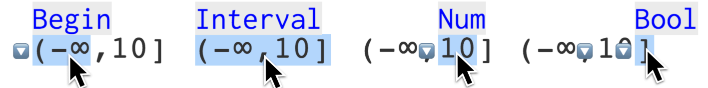

Fig. 1. Selection regions and UI elements generated by tracing the execution of the `toString` function for an interval data type.

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 data structures (such as abstract syntax trees). Liberated from raw text buffers, structure editors can use domain-specific representations for display. For example, the Barista [6] editor for Java offers rich, custom, type-specific views for mathematical and logical expressions in code. But display is easier than editing: for editing, Barista falls back to ordinary textual manipulation.

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 users 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. Our approach provides a structure editor on a domain-specific representation of a value, without the labor of writing a parser. These approaches could be combined: a structure editor could offer an optional text interface for bulk input, although our prototype does not yet.

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 custom *algebraic data types (ADTs)* in an ordinary functional language similar to Elm.[1]

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].

```
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.

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, ADTs are beginning to appear in mainstream languages: "enums" in Swift and Rust are ADTs, as are "case classes" in Scala and "discriminated unions" in Typescript.

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:

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

[1] https://elm-lang.org/. But note that, following OO and imperative languages, we use parentheses for function calls and constructor calls in TSE.

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).

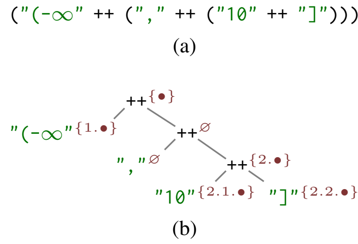

Fig. 3. A concatenation tree (two views).

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 a tree, the regions are nested tightly (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).

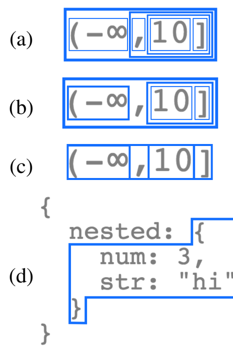

Fig. 4. String concatenations are translated into 2D regions atop the rendered string.

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, in some cases, multiple buttons would be rendered in identical locations. Such overlapping buttons are coalesced into a single button that opens a menu offering the different transformations.

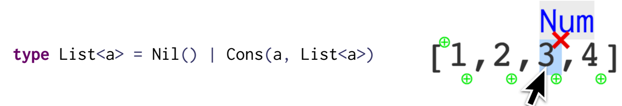

Fig. 5. Generated GUI for list `Cons(1,Cons(2,Cons(3,Cons(4,Nil()))))`.

Additional Tracing Details

In ordinary TML [3], 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`.

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.

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. While our independent implementation highlights TSE's key techniques, applying TSE to a particular application requires 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 we imagine such an interface would 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 [25], 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 will be available in an accompanying technical report.

Acknowledgments

Our thanks to Andrew McNutt, 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] M. Baudinet and D. MacQueen, “Tree Pattern Matching for ML,” 1985.