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,Cons(3,Nil())))
```

Removing the `2` means replacing the **bolded** subvalue above with its recursive child (underlined). 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 $\boxed{E \vdash e \Downarrow w}$

$$\text{[EvalVar]}\ \frac{}{E \vdash x \Downarrow E(x)} \qquad \text{[EvalDynVar]}\ \frac{x \in \{\text{“}toString''\text{, “}showsPrecFlip''\}}{E \vdash x \Downarrow \texttt{dyncall}(x)^{\{\}}} \qquad \text{[EvalFun]}\ \frac{}{E \vdash f(x).e \Downarrow ([E]\, f(x).e)^{\{\}}}$$

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

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

$$\text{[EvalCtor]}\ \frac{\overline{E \vdash e_i \Downarrow w_i}}{E \vdash C(e_1, \ldots, e_n) \Downarrow C(w_1, \ldots, w_n)^{\{\}}} \qquad \text{[EvalCase]}\ \frac{E \vdash e \Downarrow C_j(w_1, \ldots, w_n)^{p} \quad E,\, x_1 \mapsto w_1, \ldots, x_n \mapsto w_n \vdash e_j \Downarrow {v_j}^{p_j}}{E \vdash \texttt{case}\ e\ \texttt{of}\ \overline{C_i(x_1, \ldots, x_n) \to e_i} \Downarrow {v_j}^{p \cup p_j}}$$

$$\text{[EvalStr]}\ \frac{}{E \vdash s \Downarrow s^{\{\}}} \qquad \text{[EvalConcat]}\ \frac{E \vdash e_1 \Downarrow w_1 \quad E \vdash e_2 \Downarrow w_2}{E \vdash e_1 +\!\!+ e_2 \Downarrow (w_1 +\!\!+ w_2)^{\{\}}} \qquad \text{[EvalStrLen]}\ \frac{E \vdash e \Downarrow w \quad strLen(w) = n \quad allDepsDeep(w) = p}{E \vdash \texttt{strLen}(e) \Downarrow n^{p}}$$

$$\text{[EvalNum]}\ \frac{}{E \vdash n \Downarrow n^{\{\}}} \qquad \text{[EvalBinop]}\ \frac{E \vdash e_1 \Downarrow {n_1}^{p_1} \quad E \vdash e_2 \Downarrow {n_2}^{p_2} \quad n_1 \oplus n_2 = v}{E \vdash e_1 \oplus e_2 \Downarrow v^{p_1 \cup p_2}} \qquad \text{[EvalNumToStr]}\ \frac{E \vdash e \Downarrow n^{p} \quad numToStr(n) = s}{E \vdash \texttt{numToStr}(e) \Downarrow s^{p}}$$

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

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 $w_1 +\!\!+ w_2$ (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.