Tiny Structure Editors for Low, Low Prices!
(Generating GUIs from toString Functions)

Brian Hempel and Ravi Chugh
Interval(NegInf(), Before(10, True()))

$(-\infty, 10]$
Interval(NegInf(), Before(10, True()))

\((-\infty, 10]\)
Interval(NegInf(), Before(10, True()))

\((-\infty, 10]\)

But why?
Long-Term Vision
Long-Term Vision

code

program’s internal data values
Long-Term Vision

code

manipulate program’s internal data values
Long-Term Vision

code

manipulate program’s internal data values
Long-Term Vision

What about programmer's custom types?

Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)
Manipulable Visualizations for Custom Types?
Manipulable Visualizations for Custom Types?

Pointer graph?

Interval
NegInf
Before 10 True
Manipulable Visualizations for Custom Types?

Pointer graph?

Generic text? Interval(NegInf(),Before(10,True))
Manipulable Visualizations for Custom Types?

*Pointer graph?*

*Generic text?*  Interval(NegInf(),Before(10,True))
Manipulable Visualizations for Custom Types?

Pointer graph?

Generic text?

Custom vis?

Interval(NegInf(),Before(10,True))
Manipulable Visualizations for Custom Types?

Pointer graph?

Generic text?  \( \text{Interval(NegInf(),Before(10,True))} \)

Custom vis?
Manipulable Visualizations for Custom Types?

Pointer graph?

Generic text? \(\text{Interval(NegInf(),Before(10, True))}\)

Custom vis? \((-\infty, 10]\)
Manipulable Visualizations for Custom Types?

- Pointer graph?
- Generic text? \( \text{Interval(NegInf(),Before(10,True))} \)
- Custom vis?
- Custom \( \text{toString} \)? \( (-\infty,10] \)
Manipulable Visualizations for Custom Types?

Pointer graph?

Generic text? \( \text{Interval}(\text{NegInf}(), \text{Before}(10, \text{True})) \)

Custom vis?

Custom to\(\text{String}?? \) \((-\infty, 10]\)
Related Systems

Track only strings to output.

Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis

Xiaoyin Wang1, Lu Zhang2, Tao Xie1, Yingfei Xiong2, Hong Mei1
1Key Laboratory of High Confidence Software Technologies (Peking University), MOE, China
2Department of Computer Science, North Carolina State University, USA
{wangxy08,zhanglu,xiongy04,mei}@sei.pku.edu.cn, xie@cs.csc.ncsu.edu

ABSTRACT

Web applications are becoming increasingly popular. Due to the development and evolution of a web application, a typical type of tasks is to change the appearance of the web applications, such as correcting display errors, adding user interface controls, or updating hyperlinks. In order to facilitate this process, we need to be able to track changes in the presentation of a given web application. This paper presents our system for automatically determining which presentation changes occur in a web page as a result of some input action. The system is implemented as a set of tool components that work together to detect the changes in the presentation of a given web application.

1. INTRODUCTION

Recently, web applications are becoming increasingly popular due to easier access to the internet. Various researchers have developed techniques to facilitate the development and evolution of web applications, such as testing web applications (1, 3, 18), static checking for bugs in web applications (1, 5, 8), and optimization of web applications (2, 10). A new type of tool is being used to help web developers in analyzing and improving web applications. Such tools can be used to track changes in the presentation of a web application as a result of some input action. This paper presents our system for automatically determining which presentation changes occur in a web page as a result of some input action. The system is implemented as a set of tool components that work together to detect the changes in the presentation of a given web application.
Related Systems

Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis

Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, Hong Mei
Department of Computer Science, University of North Carolina at Chapel Hill, USA
Wangyx89@duke.edu, xie@cs.wisc.edu

ABSTRACT
Modern web applications are becoming increasingly popular due to their ease of use and integration into end-user applications. In this paper, we propose a framework that uses a hybrid approach to automatically detect changes in the presentation of web applications. The framework consists of three main components: a data synchronization layer, a presentation synchronization layer, and a user interaction layer. The data synchronization layer monitors changes in the data model of the web application, while the presentation synchronization layer updates the presentation of the web application in real-time. The user interaction layer allows users to interact with the web application in a more intuitive and natural way.

1. INTRODUCTION
Modern web applications are becoming increasingly popular due to their ease of use and integration into end-user applications. In this paper, we propose a framework that uses a hybrid approach to automatically detect changes in the presentation of web applications. The framework consists of three main components: a data synchronization layer, a presentation synchronization layer, and a user interaction layer. The data synchronization layer monitors changes in the data model of the web application, while the presentation synchronization layer updates the presentation of the web application in real-time. The user interaction layer allows users to interact with the web application in a more intuitive and natural way.

Related Systems

Schuster & Flanagan (2016)

Wang et al. (2012)

Track only strings to output.

But we want to track any value of interest to output.
Two Key Ideas

1. Generic dependency tracing.
Abstract

Provenance is an increasing concern due to the ongoing revolution in sharing and processing scientific data on the Web and in other computer systems. It is proposed that many computer systems will need to become provenance-aware in order to provide satisfactory accountability, reproducibility, and trust for scientific or other high-value data. To date, there is not a consensus concerning appropriate formal models or security properties for provenance. In previous work, we introduced a formal framework for provenance security and proposed formal definitions of properties called disclosure and obfuscation.

In this article, we study refined notions of positive and negative disclosure and obfuscation in a concrete setting, that of a general-purpose programming language. Previous models of provenance have focused on special-purpose languages such as workflows and database queries. We consider a higher-order, functional language with sums, products, and recursive types and functions, and equip it with a tracing semantics in which traces themselves can be replayed as computations. We present an annotation-propagation framework that supports many provenance views over traces, including standard forms of provenance studied previously. We investigate some relationships among provenance views and develop some partial solutions to the disclosure and obfuscation problems, including correct algorithms for disclosure and positive obfuscation based on trace slicing.

1 Introduction

Provenance, or meta-information about the origin, history, or derivation of an object, is now recognized as a central challenge in establishing trust and providing security in computer systems, particularly on the Web. Essentially,
Two Key Ideas

1. Generic dependency tracing from TML.
Two Key Ideas

1. Generic dependency tracing from TML.

2. To keep track of substrings: defer string concatenation.
Two Key Ideas

1. Generic dependency tracing from TML.

2. To keep track of substrings: defer string concatenation.

   "a" ++ "b" ++ "c" ↓ "abc"
Two Key Ideas

1. Generic dependency tracing from TML.

2. To keep track of substrings: defer string concatenation.

```
"a" ++ "b" ++ "c" \downarrow "abc"
```
Two Key Ideas

1. **Generic dependency tracing from TML.**

2. **To keep track of substrings: defer string concatenation.**

```
"a" ++ "b" ++ "c" ↓ "abc"

"a" ++ "b" ++ "c" ↓ (("a" ++ "b") ++ "c")
```
Interval(NegInf(), Before(10, True))
Interval(NegInf(), Before(10, True))

Tag subvalues with identifiers.

Interval(NegInf({1.}, Before(10{2.1.}, True{2.2.}){2.}){·})
Based on the value's type, the appropriate final actions are assigned to the 2D regions.

Interval(NegInf(), Before(10, True))

Tag subvalues with identifiers.

Interval(NegInf(){1.●},Before(10{2.1.●},True{2.2.●}){2.●}){●}

Call toString using TML.

("(-∞" ++ ("," ++ ("10" ++ "]")))
Interval(NegInf(), Before(10, True))

Tag subvalues with identifiers.

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

Call toString using TML.

"(-\infty"

"","

"10"

"]"
Based on the value's type, the appropriate constructors are first tagged with that traces dependency provenance, following Transparencies. Finally, actions are assigned to the 2D regions.

In the final display, selection regions are overlaid on top of the rendered string, the deepest (equivalently, smallest) selection regions, shown in Figure 10c. Selection is curried over the rendered string, the deepest (equivalently, smallest) selection regions are overlaid on top of the string (b) and the user may interact with these regions (c) to manipulate the selections and actions.

Since concatenation produces a new, flattened string. Instead, the concatenation evaluator propagates the dependency tags. Additionally, each substring and each concatenation carves a set of projection paths, relating parts of the string to the base value.

The type definitions produce a boolean indicating whether the boundary is or is not infinite, the bound records the finite boundary number and the interval type.

\[
\text{Interval}(\text{NegInf}(), \text{Before}(10, \text{True}))
\]

- Tag subvalues with identifiers.
- Call \(\text{toString} \) using TML.

\[
\text{Interval}(\text{NegInf}()\{1.\}, \text{Before}(10\{2.1.\}, \text{True}\{2.2.\})\{2.\})\{\}
\]

\[
"(-\infty)\{1.\}++\{\}\n++\emptyset\n++\{2.\}\n"
\]

\[
",",\emptyset\n++\{2.\}\n"
\]

\[
"10\{2.1.\} "\]"\{2.2.\}
\]
Interval(NegInf(), Before(10, True))

Tag subvalues with identifiers.

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

Call toString using TML.

"(-\infty)\{1.\}

++\{\}

"\","

++\{2.\}

"10"\{2.1.\} "\]"\{2.2.\}

Assign spatial regions.

\((-\infty,10]\) \rightarrow (\(-\infty,10]\)
Interval(NegInf(), Before(10, True))

- Tag subvalues with identifiers.
- Call toString using TML.
- Assign spacial regions.
- Assign action buttons (√ swaps variants).

Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)
type Begin = NegInf() | After(Num, Bool)

(type End = Before(Num, Bool) | Inf()

type Interval = Interval(Begin, End)

Interval(NegInf(), Before(10, True()))

(−∞, 10]
type Begin    = NegInf()     | After(Num, Bool)
type End      = Before(Num, Bool) | Inf()
type Interval = Interval(Begin, End)

Interval(NegInf(), Before(10, True()))

\((-\infty, 10]\)
\texttt{type List\textlangle a\textrangle = Nil() | Cons(a, List\textlangle a\textrangle)}

\begin{verbatim}
Cons(1, Cons(2, Cons(3, Nil())))
\end{verbatim}

[1, 2, 3]
Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)

```haskell
type List<a> = Nil() | Cons(a, List<a>)
```

Cons(1, Cons(2, Cons(3, Nil())))

[1, 2, 3]
## Case Studies

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>%Selectable Subvalues</th>
<th>%Selectable Items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tiny Structure Editors for Low, Low Prices! (Generating GUIs from `toString` Functions)
## Case Studies

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>%Selectable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td></td>
</tr>
<tr>
<td>JSON (multiline)</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td></td>
</tr>
<tr>
<td>List (&quot;[]&quot; in base case)</td>
<td></td>
</tr>
<tr>
<td>List (via join)</td>
<td></td>
</tr>
<tr>
<td>List (via different join)</td>
<td></td>
</tr>
<tr>
<td>Tree (S-exp)</td>
<td></td>
</tr>
<tr>
<td>Tree (indented hierarchy)</td>
<td></td>
</tr>
</tbody>
</table>

Tiny Structure Editors for Low, Low Prices! (Generating GUIs from `toString` Functions)
# Case Studies

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>%Selectable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Subvalues</td>
</tr>
<tr>
<td>Interval</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td></td>
</tr>
<tr>
<td>JSON (multiline)</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td></td>
</tr>
<tr>
<td>List (&quot;]&quot;) in base case</td>
<td></td>
</tr>
<tr>
<td>List (via join)</td>
<td></td>
</tr>
<tr>
<td>List (via different join)</td>
<td></td>
</tr>
<tr>
<td>Tree (S-exp)</td>
<td></td>
</tr>
<tr>
<td>Tree (indented hierarchy)</td>
<td></td>
</tr>
<tr>
<td>Pair [22]</td>
<td></td>
</tr>
<tr>
<td>List [22]</td>
<td></td>
</tr>
<tr>
<td>ADT (recursive) [22]</td>
<td></td>
</tr>
<tr>
<td>Record [22]</td>
<td></td>
</tr>
<tr>
<td>Set [13]</td>
<td></td>
</tr>
</tbody>
</table>
Case Studies

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>%Selectable</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval</td>
<td>80% (4/5)</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>100% (4/4)</td>
<td></td>
</tr>
<tr>
<td>JSON (multiline)</td>
<td>33% (14/43)</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td>86% (6/7)</td>
<td></td>
</tr>
<tr>
<td>List (&quot;[&quot; in base case)</td>
<td>100% (7/7)</td>
<td></td>
</tr>
<tr>
<td>List (via join)</td>
<td>71% (5/7)</td>
<td></td>
</tr>
<tr>
<td>List (via different join)</td>
<td>86% (6/7)</td>
<td></td>
</tr>
<tr>
<td>Tree (S-exp)</td>
<td>53% (10/19)</td>
<td></td>
</tr>
<tr>
<td>Tree (indented hierarchy)</td>
<td>21% (4/19)</td>
<td></td>
</tr>
<tr>
<td>Pair [22]</td>
<td>100% (3/3)</td>
<td></td>
</tr>
<tr>
<td>List [22]</td>
<td>100% (7/7)</td>
<td></td>
</tr>
<tr>
<td>ADT (recursive) [22]</td>
<td>100% (4/4)</td>
<td></td>
</tr>
<tr>
<td>Record [22]</td>
<td>100% (9/9)</td>
<td></td>
</tr>
<tr>
<td>Set [13]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Case Studies

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>%Selectable Subvalues</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval</td>
<td>80% (4/5)</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>100% (4/4)</td>
<td></td>
</tr>
<tr>
<td>JSON (multiline)</td>
<td>33% (14/43)</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td>86% (6/7)</td>
<td>100% (3/3)</td>
</tr>
<tr>
<td>List (&quot;[]&quot; in base case)</td>
<td>100% (7/7)</td>
<td>100% (3/3)</td>
</tr>
<tr>
<td>List (via join)</td>
<td>71% (5/7)</td>
<td>100% (3/3)</td>
</tr>
<tr>
<td>List (via different join)</td>
<td>86% (6/7)</td>
<td>100% (3/3)</td>
</tr>
<tr>
<td>Tree (S-exp)</td>
<td>53% (10/19)</td>
<td>100% (5/5)</td>
</tr>
<tr>
<td>Tree (indented hierarchy)</td>
<td>21% (4/19)</td>
<td>100% (5/5)</td>
</tr>
<tr>
<td>Pair [22]</td>
<td>100% (3/3)</td>
<td></td>
</tr>
<tr>
<td>List [22]</td>
<td>100% (7/7)</td>
<td>100% (3/3)</td>
</tr>
<tr>
<td>ADT (recursive) [22]</td>
<td>100% (4/4)</td>
<td></td>
</tr>
<tr>
<td>Record [22]</td>
<td>100% (9/9)</td>
<td></td>
</tr>
<tr>
<td>Set [13]</td>
<td>100% (4/4)</td>
<td></td>
</tr>
</tbody>
</table>
TSE Limitations
TSE Limitations

• Standalone prototype—needs to be adapted to some concrete setting.
TSE Limitations

• Standalone prototype—needs to be adapted to some concrete setting.

• Occlusion.
TSE Limitations

- Standalone prototype—needs to be adapted to some concrete setting.

- Occlusion.

- Insert/remove only works well for lists.
Tiny Structure Editors for Low, Low Prices!

Begin $(-\infty, 10]$  Interval $(-\infty, 10]$  Num $(-\infty, 10]$  Bool $(\infty, 10]$

Tiny Structure Editors for Low, Low Prices! (Generating GUls from toString Functions)
Tiny Structure Editors for Low, Low Prices!

Use generic dependency tracing + deferred concatenation to generate tiny structure editors from `toString` functions.

```
Begin (-∞, 10]
Interval (-∞, 10]
Num (−∞, 10] (−∞, 10]
Bool
```
Tiny Structure Editors for Low, Low Prices!

Use generic dependency tracing + deferred concatenation to generate tiny structure editors from `toString` functions.

⇒ Custom, manipulable visualizations for user-defined types.
Use generic dependency tracing + deferred concatenation to generate tiny structure editors from `toString` functions.

⇒ Custom, manipulable visualizations for user-defined types.
Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)
Expressions $e$ ::= $x \mid f(x).e \mid e_1(e_2) \\
| C(e_1, \ldots, e_n) \\
| \text{case } e \text{ of } C_i(x_1, \ldots, x_n) \rightarrow e_i \\
| s \mid e_1 ++ e_2 \mid \text{strLen}(e) \\
| n \mid e_1 \oplus e_2 \mid \text{numToStr}(e) \\
| \text{basedOn}(e_d, e)$

Projection Paths $\pi$ ::= $\bullet \mid i.\pi$

(Tagged) Values $w$ ::= $v\{\pi_1, \ldots, \pi_n\}$

(Untagged) Pre-Values $v$ ::= $[E] f(x).e \mid C(w_1, \ldots, w_n) \\
| s \mid w_1 ++ w_2 \mid n \\
| \text{dynCall}(f)$

Tagged Environments $E$ ::= $\_ \mid E, x \mapsto w$

Evaluation with Dependency Provenance

\[ E \vdash e \downarrow w \]

[EvalCase]

\[
E \vdash e \downarrow C_j(w_1, \ldots, w_n)^p \\
E, x_1 \mapsto w_1, \ldots, x_n \mapsto w_n \vdash e_j \downarrow v_j^{p_j} \\
E \vdash \text{case } e \text{ of } C_i(x_1, \ldots, x_n) \rightarrow e_i \downarrow v_j^{p \cup p_j}
\]
Figure 9: ADT and Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)

```haskell
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() -> "(-\infty"
    After(num, isClosed) ->
        (if isClosed then "[" else "]") ++ toString(num)

toString : End -> String
toString(end) = case end of
    Inf() -> "\infty)"
    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))
```
Based on the value’s type, the appropriate

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

toString : End -> String
toString(end) = case end of
    Inf() -> "]\infty)"
    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))

Algebraic data types (ADTs)
express variants a.k.a. constructors
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))
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() -> "(-\infty"
    After(num, isClosed) ->
        (if isClosed then "[" else "]") ++ toString(num)

toString : End -> String
toString(end) = case end of
    Inf() -> "\infty")
    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))
Based on the value’s type, the appropriate are first tagged with that traces dependency provenance, following Transpar-

Finally actions are assigned to the 2D regions. Spatial regions over the rendered string are computed, and editors proceeds in three steps. The tracing evaluator re-

5.2 Algorithm

In the which define the handling of alternative variants, as shown

Custom interval data type. The lower bound of an interval

Figure 9: ADT and TSE’s automatic algorithm for generating tiny structure editors for low, low prices! (Generating GUIs from toString functions)

```
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))
```
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))
The value of interest and its subvalues that traces dependency provenance, following Transparency, finally, actions are assigned to the 2D regions. Spatial regions over the rendered string are computed, and substrings to portions of the original value, then 2D editors proceeds in three steps. The tracing evaluator re...

interval type. The lower bound of an interval is included in the interval (is or is not finite, the bound records the finite boundary number and is negative infinity). This constructor to create an interval value representing upper boundaries (is or is not finite, the bound records the finite boundary number and is negative infinity). The type de...

TSE utilizes a custom evaluator that acts as a function to "pattern matching" in these constructors. The last line of Figure 9 uses case splits (structure editors). The following definitions for a custom interval data type. The lower bound of an interval is included in the interval (is or is not finite, the bound records the finite boundary number and is negative infinity). The type de...

Figure 10: ADT and Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)

```plaintext
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() -> "(-\infty"
    After(num, isClosed) ->
        (if isClosed then "[" else "]") ++ toString(num)

toString : End -> String
toString(end) = case end of
    Inf() -> "\infty")
    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))
```

Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)
Finally, actions are assigned to the 2D regions.

Editors proceed in three steps. The tracing evaluator requires inputs (i) which define the handling of alternative variants, as shown in Figure 9.

After that traces dependency provenance, following Transparency Selection Editors (TSE)’s automatic algorithm for generating tiny structure editors for low, low prices! (Generating GUIs from toString Functions)

```
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, False))
```
**Type Definitions and String Transformation Functions**

- `type Begin = NegInf()`
- `type End = Before(Num, Bool)`
- `type Interval = Interval(Begin, End)`

```haskell
toString : Begin -> String
toString(begin) = case begin of
    NegInf() -> "(-\infty"
    After(num, isClosed) ->
        (if isClosed then "[" else "]") ++ toString(num)

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

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

**Example Usage**

```haskell
valueOfInterest : Interval
valueOfInterest = Interval(NegInf(), Before(10, True))
```
Based on the value’s type, the appropriate projection paths are first tagged with editors proceeds in three steps. The tracing evaluator re-

\[
\text{valueOfInterest} : \text{Interval} = \text{Interval}(\text{NegInf()}, \text{Before}(10, \text{False}))
\]

\[
\text{ToString : Begin} \rightarrow \text{String}\\
\text{ToString(begin)} = \text{case begin of}\\
\quad \text{NegInf()} \rightarrow "(-\infty)"\\
\quad \text{After(num, isClosed)} \rightarrow (\text{if isClosed then }[" else "](\)) ++ \text{toString(num)}
\]

\[
\text{ToString : End} \rightarrow \text{String}\\
\text{ToString(end)} = \text{case end of}\\
\quad \text{Inf()} \rightarrow "(\infty)"\\
\quad \text{Before(num, isClosed)} \rightarrow \text{toString(num)} ++ (\text{if isClosed then }"]" else ")"
\]

\[
\text{ToString : Interval} \rightarrow \text{String}\\
\text{ToString(interval)} = \text{case interval of}\\
\quad \text{Interval(begin, end)} \rightarrow \text{toString(begin)} ++ "," ++ \text{toString(end)}
\]
Based on the value's type, the appropriate
projection paths are first tagged with
that traces dependency provenance, following Transpar-
ency Tracing finally actions are assigned to the 2D regions.
Spatial Regions over the rendered string are computed, and
lates substrings to portions of the original value, then 2D
editors proceeds in three steps. The tracing evaluator re-
5.2 Algorithm which define the handling of alternative variants, as shown
scribing upper boundaries (is or is not
included in the interval (is or is not
finite, the bound records the finite boundary number and
is negative infinity (NegInf)
interval type.

Figure 9: ADT and

Begin Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)

\[
\text{Interval} = \text{Interval}(\text{Begin}, \text{End})
\]

\[
\begin{align*}
toString : \text{Begin} & \rightarrow \text{String} \\
toString(\text{begin}) &= \text{case} \begin{cases} \\
\text{NegInf}() & \rightarrow "(-\infty)" \\
\text{After}(\text{num, isClosed}) & \rightarrow \\
(\text{if isClosed then } "[" \text{ else } "]") & ++ \text{toString}(\text{num})
\end{cases} \\
toString : \text{End} & \rightarrow \text{String} \\
toString(\text{end}) &= \text{case} \begin{cases} \\
\text{Inf}() & \rightarrow "\infty)" \\
\text{Before}(\text{num, isClosed}) & \rightarrow \\
\text{toString}(\text{num}) & ++ (\text{if isClosed then } "]" \text{ else "]"})
\end{cases} \\
toString : \text{Interval} & \rightarrow \text{String} \\
toString(\text{interval}) &= \text{case} \begin{cases} \\
\text{Interval}(\text{begin, end}) & \rightarrow \\
\text{toString}(\text{begin}) & ++ "," & ++ \text{toString}(\text{end})
\end{cases}
\end{align*}
\]

\[
\text{valueOfInterest : Interval} \\
\text{valueOfInterest} = \text{Interval}(\text{NegInf}(), \text{Before}(10, \text{True}))
\]
**Type Definitions and Implementations**

```haskell
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() -> "(-\infty"
    After(num, isClosed) ->
      (if isClosed then "[" else "]") ++ toString(num)

toString : End -> String
  toString(end) = case end of
    Inf() -> "\infty"
    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, False))
```

---

**Dependency Tracing**

Finally actions are assigned to the 2D regions. Editors proceed in three steps. The tracing evaluator recursively explores three: (a) case splits, switch statements) or finite (−\infty), True), Namely item insertion, item removal, and constructor swapping. For the interval example, there are four possible constructs to consider: (1) a base value (−\infty, 10), namely item insertion, item removal, and constructor swapping. Namely item insertion, item removal, and constructor swapping.

---

**Instrumented Execution**

Instrumented execution with deferred concatenation produces a string (a1) with each substring associated with parts of the value of interest (a2). In the rendered UI, regions are associated with corresponding locations over the rendered string, the deepest (equivalently, smallest) substring has been selected (Figure 10b). Selections and actions explore three: (a) base value, (b) spatial region of the associated substring (Figure 10b). After execution completes (Figure 10a), the tracing evaluator propagates the dependency tags. Additionally, the UI widgets are overlaid on top of the rendered string. To generate the selection regions, the string concatenation binary tree is translated into a binary tree of substrings when parts of the value of interest (Figure 10a). The type definitions for a custom Instrumented execution with deferred concatenation produces a string (a1) with each substring associated with parts of the value of interest (a2). In the rendered UI, regions are associated with corresponding locations over the rendered string, the deepest (equivalently, smallest) substring has been selected (Figure 10b). Selections and actions explore three: (a) base value, (b) spatial region of the associated substring (Figure 10b). After execution completes (Figure 10a), the tracing evaluator propagates the dependency tags. Additionally, the UI widgets are overlaid on top of the rendered string. To generate the selection regions, the string concatenation binary tree is translated into a binary tree of substrings when parts of the value of interest (Figure 10a). The type definitions for a custom
Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)