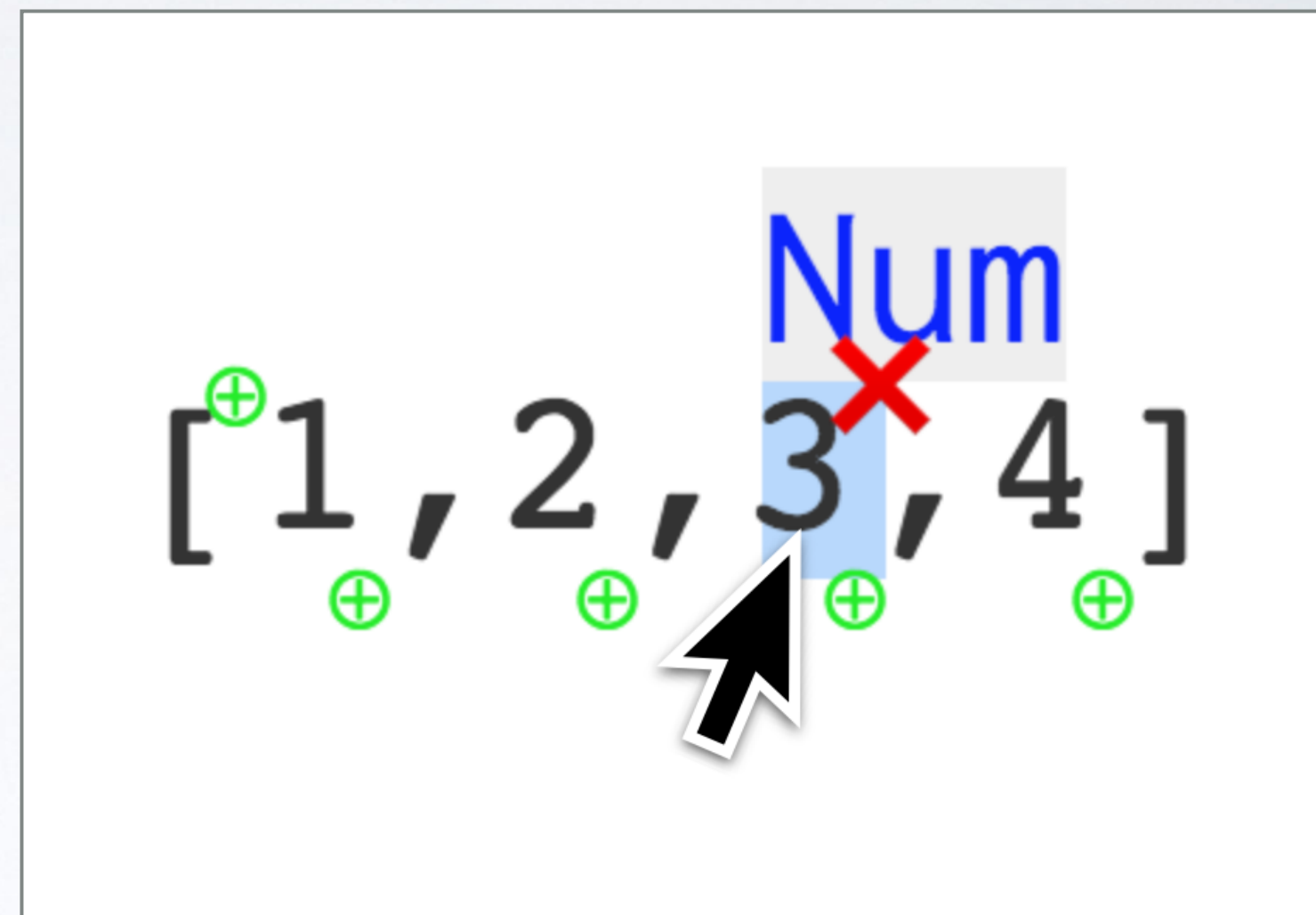
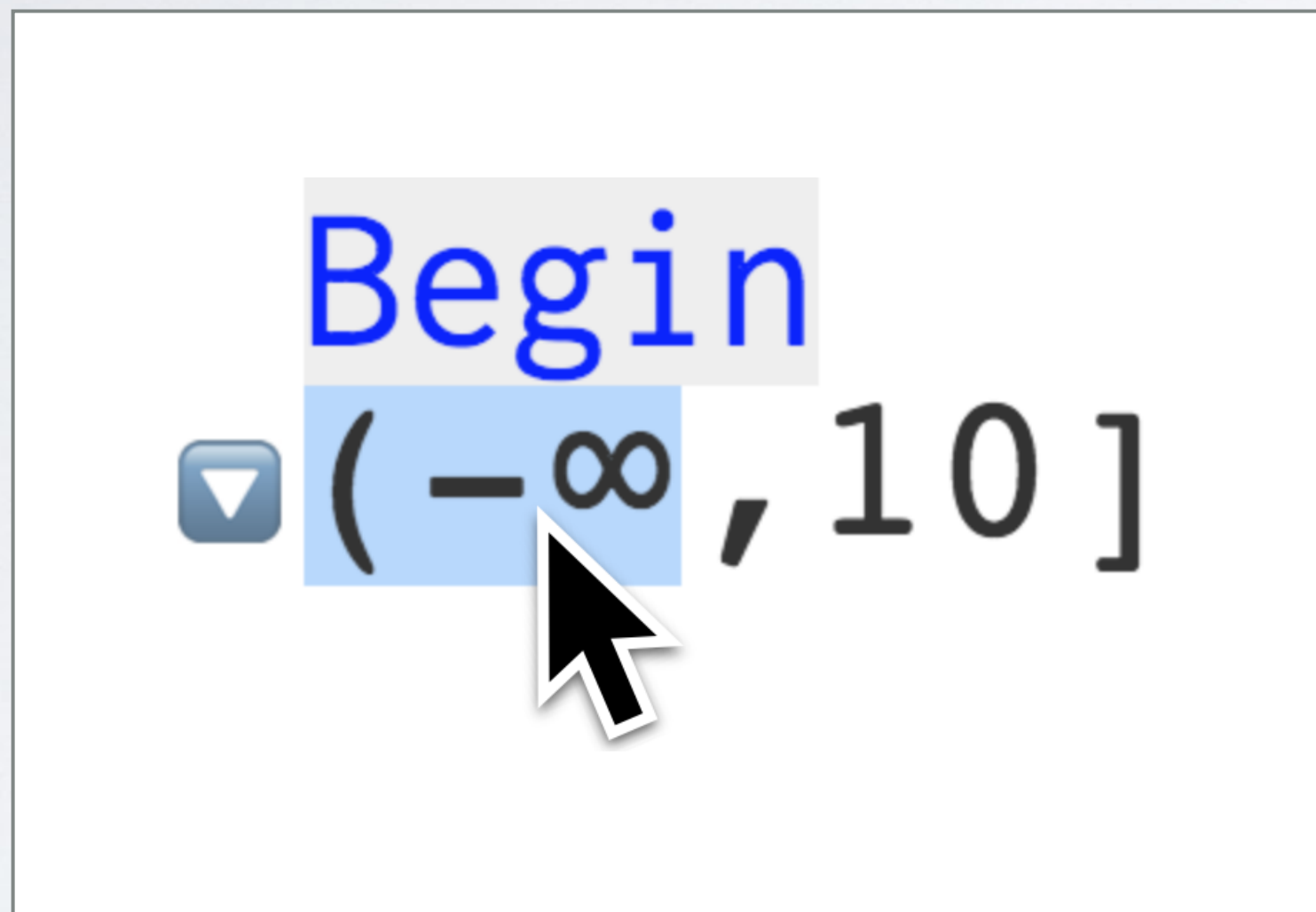


# Tiny Structure Editors for Low, Low Prices!

(Generating GUIs from toString Functions)



Brian Hempel and Ravi Chugh



THE UNIVERSITY OF  
CHICAGO

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

```
 $(-\infty, 10]$ 
```

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

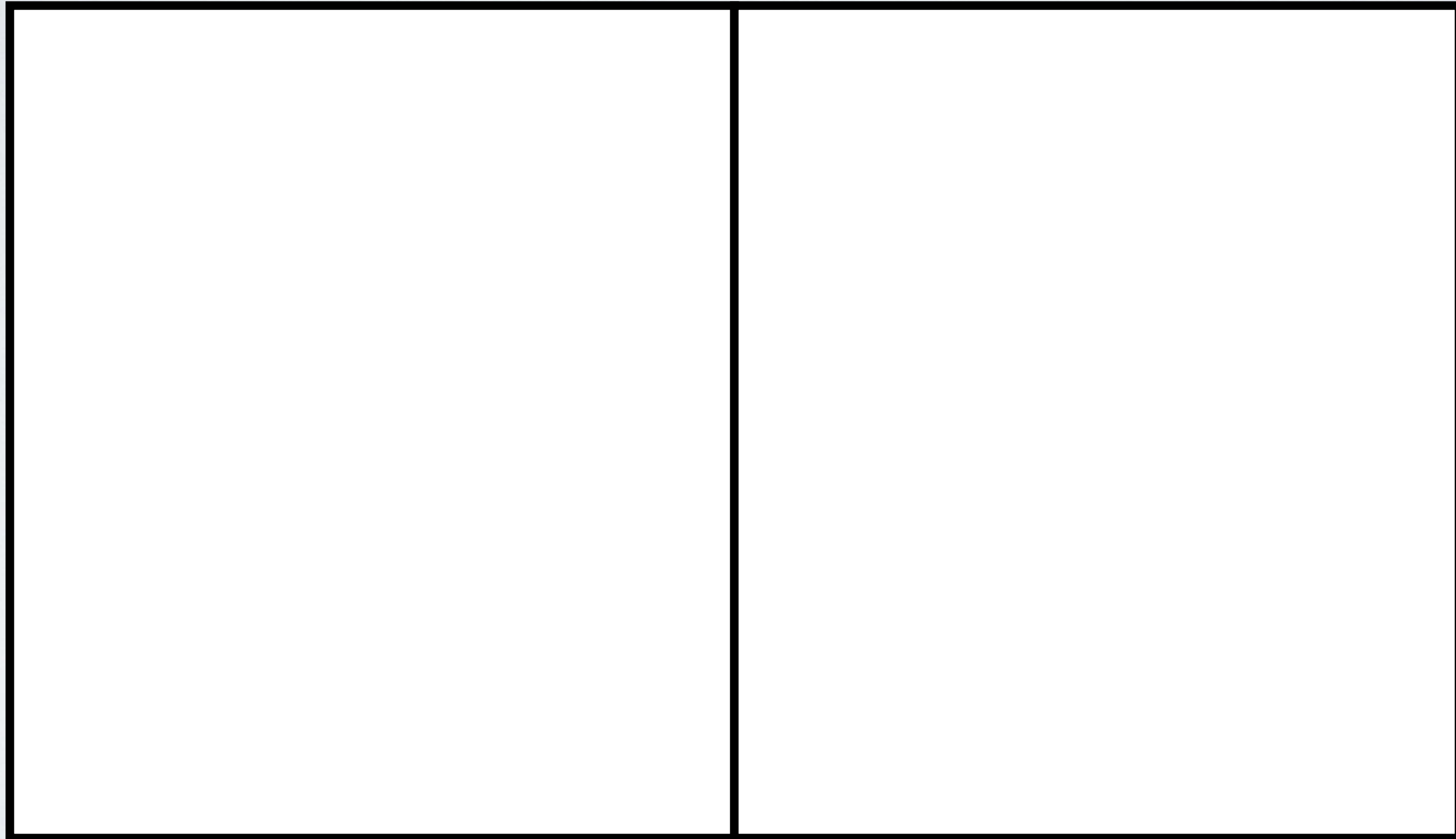
```
 $(-\infty, 10]$ 
```

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

```
( $-\infty$ , 10]
```

**But why?**

# Long-Term Vision



# Long-Term Vision

code

program's  
internal  
data values

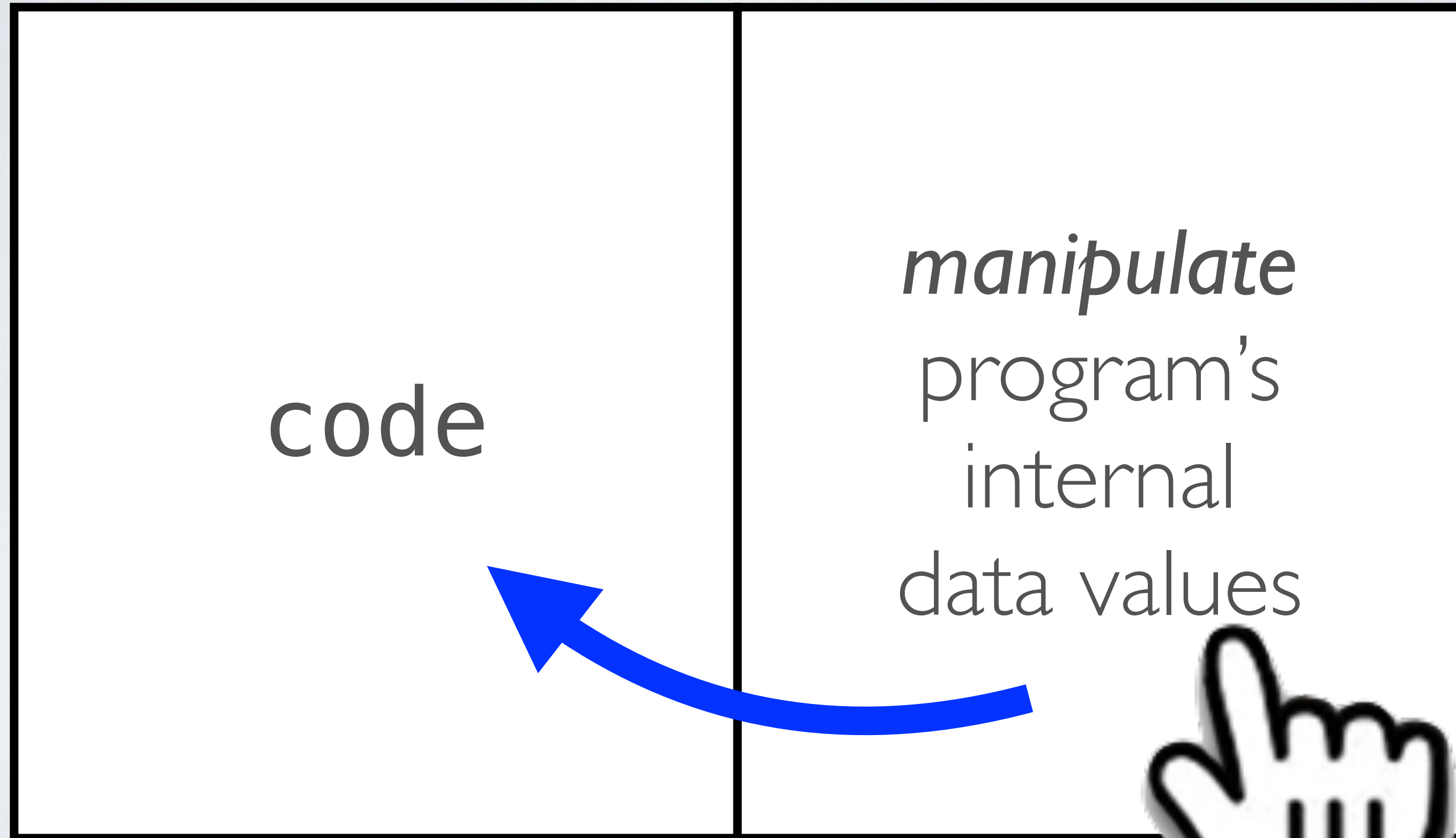
# Long-Term Vision

code

*manipulate*  
program's  
internal  
data values

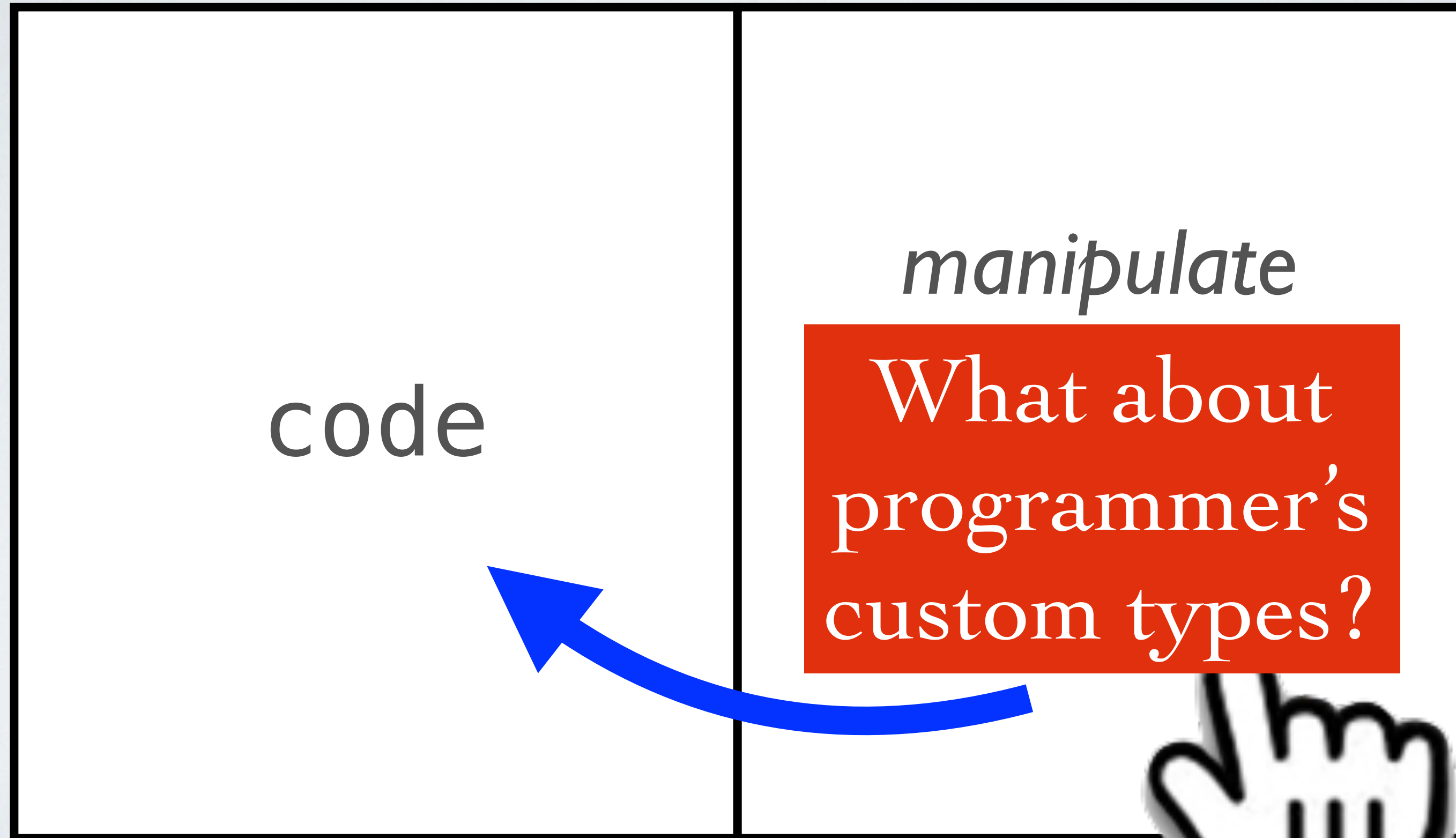


# Long-Term Vision





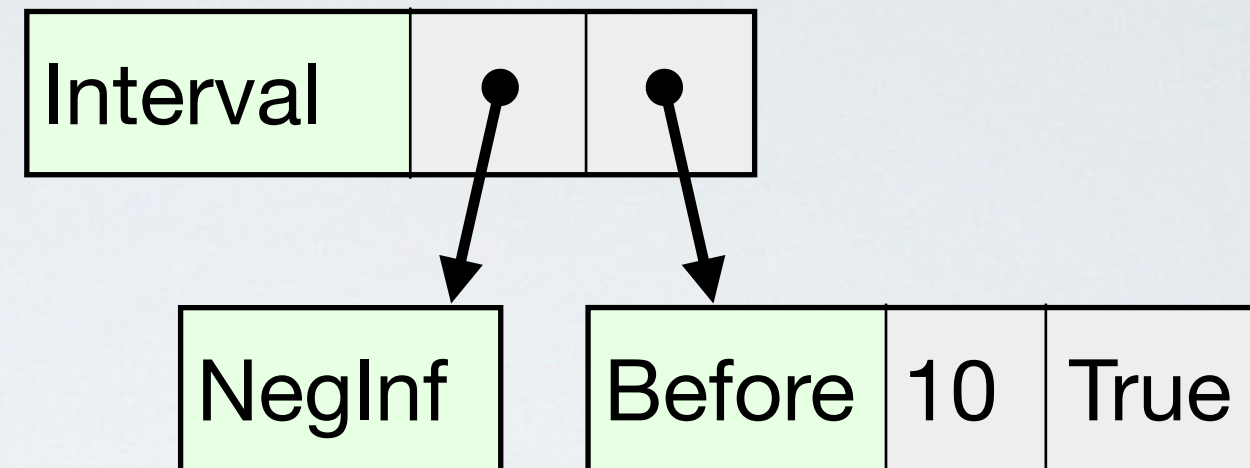
# Long-Term Vision



# Manipulable Visualizations for Custom Types?

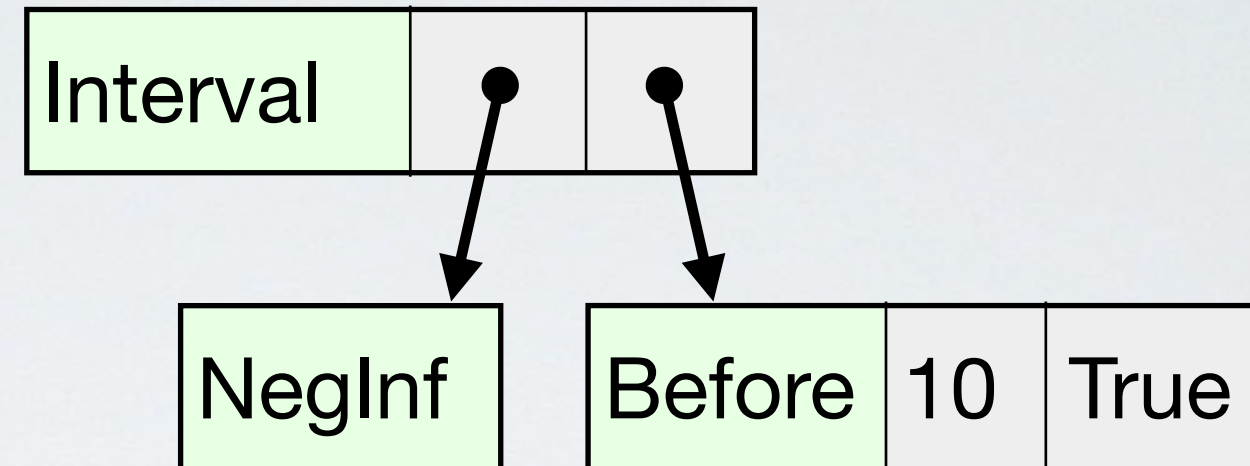
# Manipulable Visualizations for Custom Types?

Pointer graph?



# Manipulable Visualizations for Custom Types?

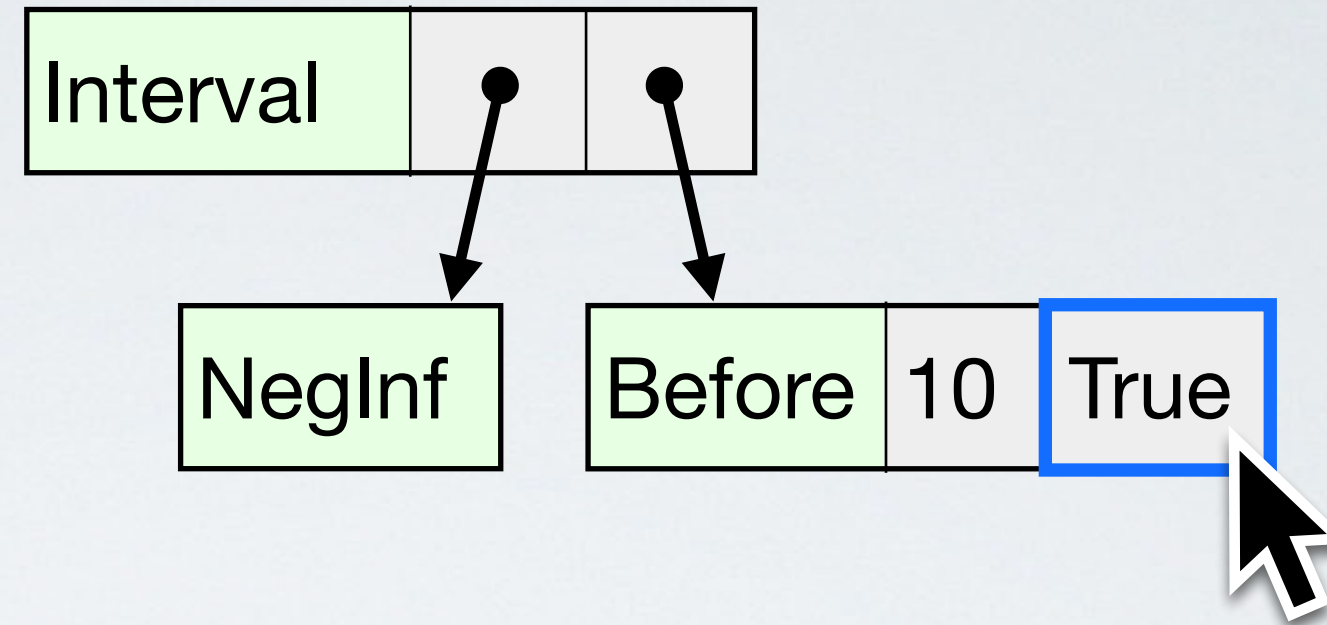
Pointer graph?



Generic text? `Interval(NegInf(), Before(10, True))`

# Manipulable Visualizations for Custom Types?

Pointer graph?

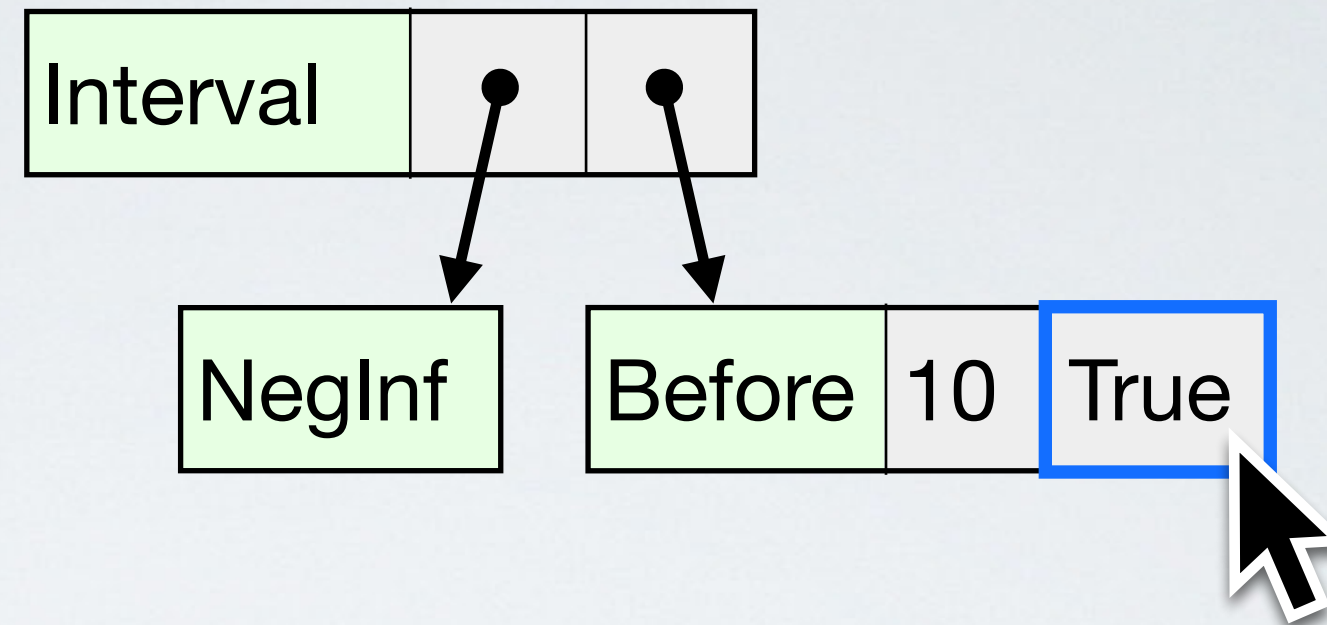


Generic text?

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

# Manipulable Visualizations for Custom Types?

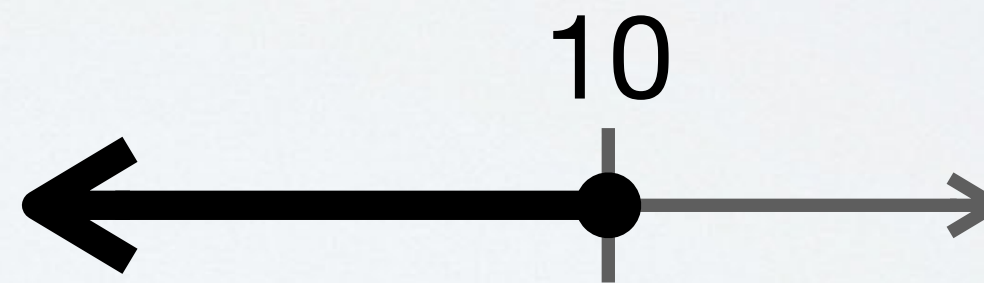
Pointer graph?



Generic text?

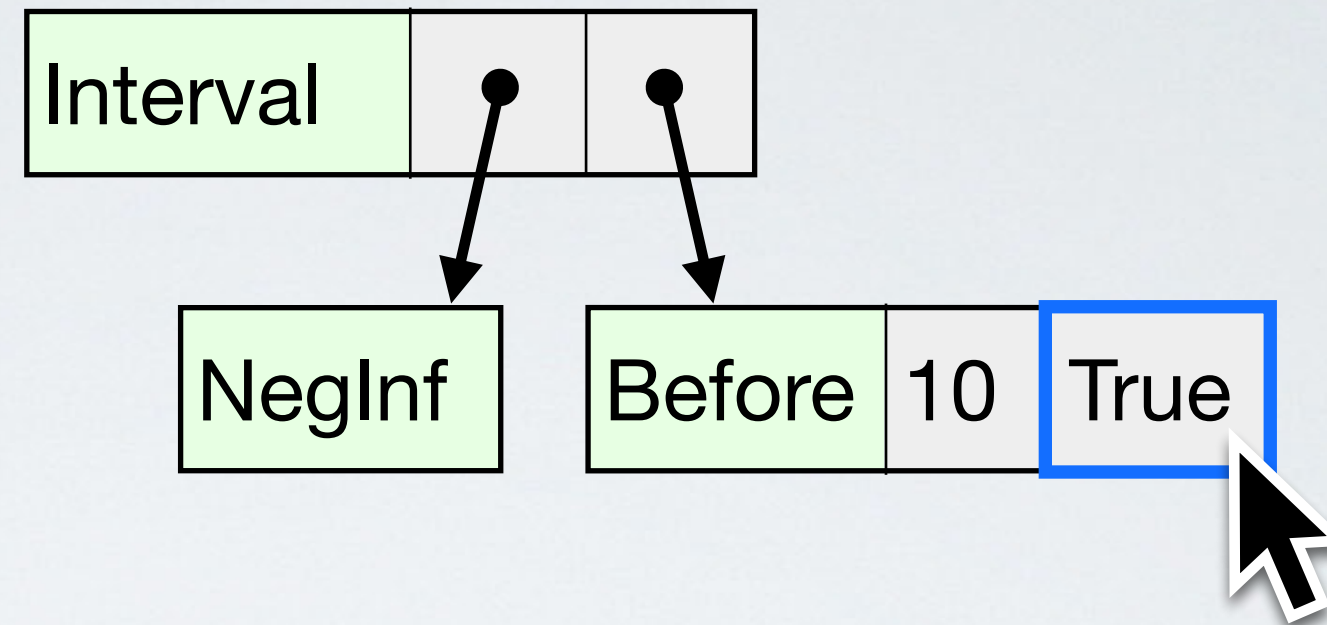
`Interval(NegInf(), Before(10, True))`

Custom vis?



# Manipulable Visualizations for Custom Types?

Pointer graph?



Generic text?

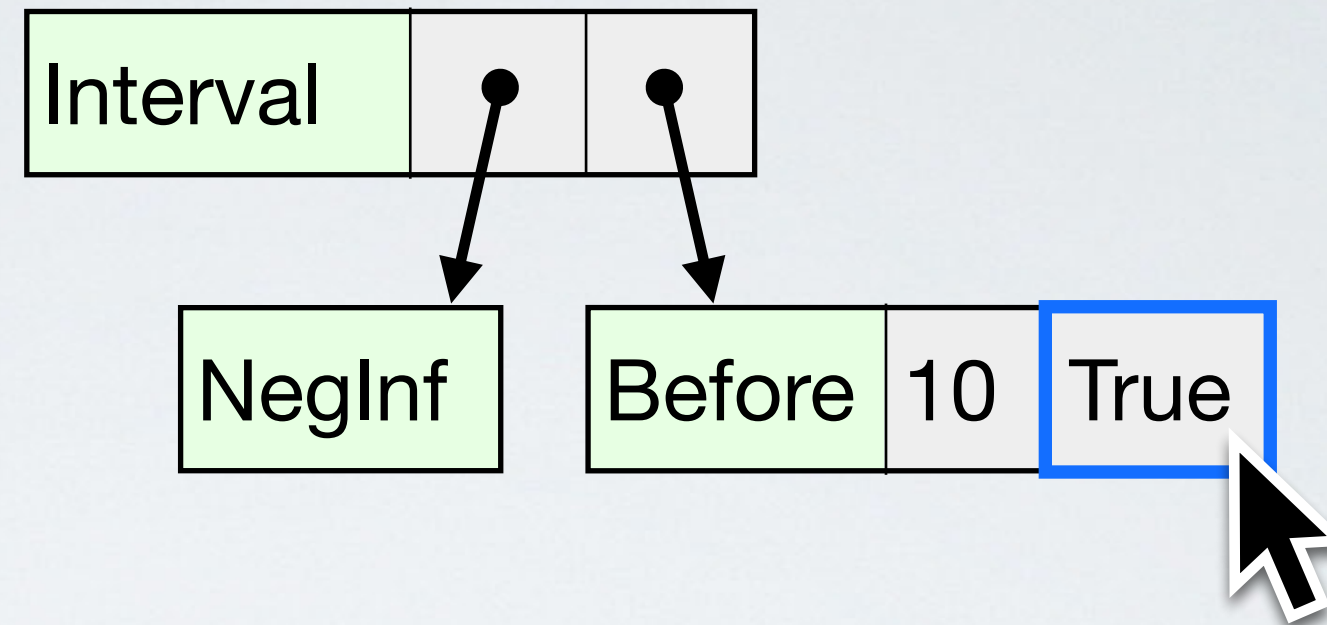
`Interval(NegInf(), Before(10, True))`

Custom vis?



# Manipulable Visualizations for Custom Types?

Pointer graph?



Generic text?

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

Custom vis?



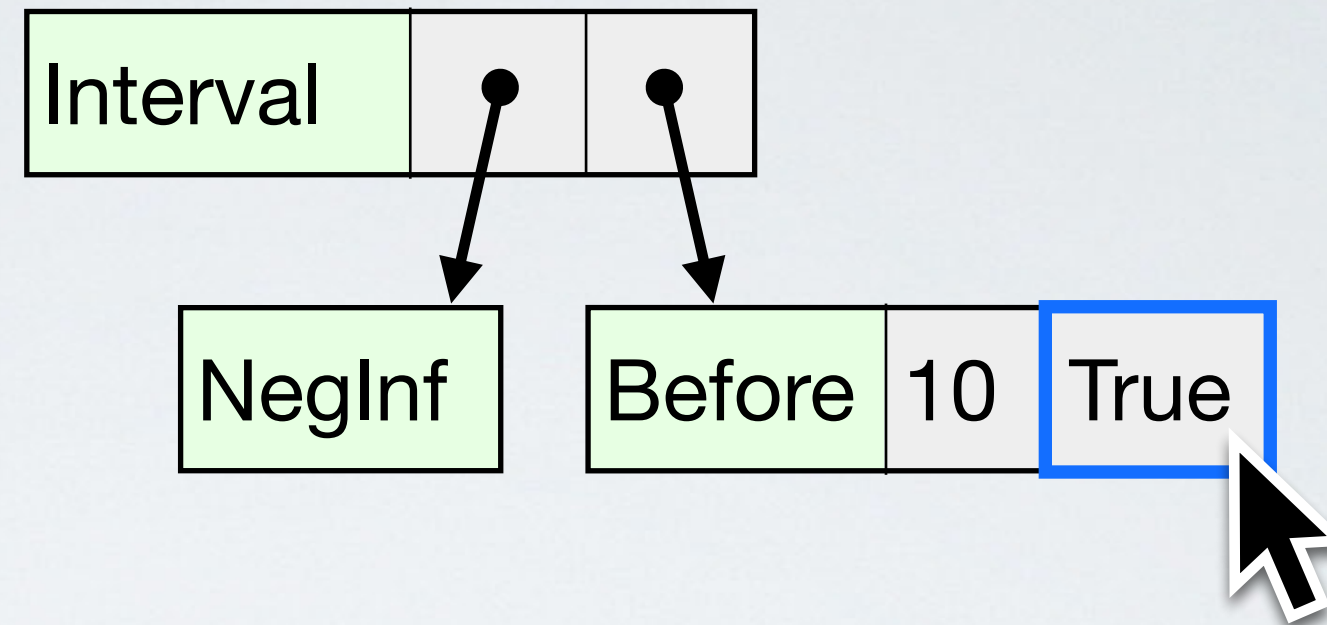
Custom toString?

`$(-\infty, 10]$`



# Manipulable Visualizations for Custom Types?

Pointer graph?



Generic text?

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

Custom vis?

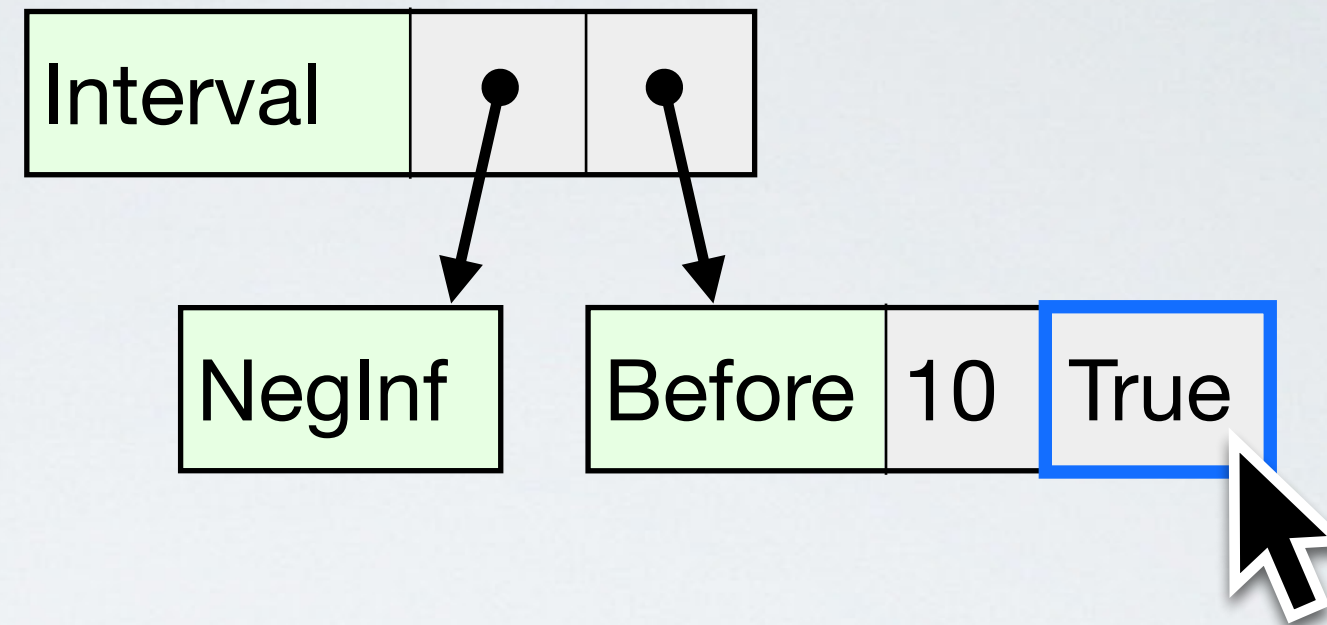


Custom toString?

`(-∞, 10]`

# Manipulable Visualizations for Custom Types?

Pointer graph?



Generic text?

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

Custom vis?



Custom toString?

`(-∞, 10]`

# Related Systems

Track only strings to output.

## Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis

Xiaoyin Wang<sup>1</sup>, Lu Zhang<sup>1</sup>; Tao Xie<sup>2</sup>, Yingfei Xiong<sup>1</sup>, Hong Mei<sup>1</sup>  
<sup>1</sup>Key Laboratory of High Confidence Software Technologies (Peking University), MOE, China  
<sup>2</sup>Department of Computer Science, North Carolina State University, USA  
{wangxy06,zhanglu,xiongyf04,meih}@sei.pku.edu.cn, xie@csc.ncsu.edu

### ABSTRACT

Web applications are becoming increasingly popular nowadays. During the development and evolution of a web application, a typical type of tasks is to change the presentation of the web application, such as correcting display errors, adding user-interface controls, or changing appearance styles. To change the presentation of a static web page, developers are able to modify the HTML text of the web page using a graphical web-page editor. However, to change the presentation of a dynamic web application, developers need a graphical web-page editor to directly modify generated web pages, developers need to modify the code that generates the web pages.

### 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 [4, 3, 19], static checking for bugs in web applications [10, 33], and refactoring web applications [31, 18]. A typical type of daily tasks during the development and evolution of web applications is presentation changes, which are modifications made to change the appearance of web pages. Typical presentation changes in web applications include correction of display errors, adding user-interface controls, etc. According to our investigation of 600 bug reports from three

Wang et al. (2012)

The screenshot shows a web development tool interface. On the left, the 'Source' panel displays JavaScript code for a 'keyup' event handler and a 'render' function. The 'keyup' function updates a state variable 'str' with the value of the event target. The 'render' function returns an HTML snippet containing an input field with the value of 'str' and an 'onkeyup' event listener. On the right, the 'Time Control' panel includes a 'Version' slider set to 1 and a 'State' slider set to 1. Below these are tabs for 'Output', 'HTML', and 'State'. The 'Output' tab is active, showing the text 'Weird, my keyboard just switched to Chinese.' with a small '1' next to it. The 'HTML' tab shows the rendered HTML, and the 'State' tab shows the current state of the application.

Schuster & Flanagan (2016)

# Related Systems

Track only strings to output.

## Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis

Xiaoyin Wang<sup>1</sup>, Lu Zhang<sup>1</sup>; Tao Xie<sup>2</sup>, Yingfei Xiong<sup>1</sup>, Hong Mei<sup>1</sup>  
<sup>1</sup>Key Laboratory of High Confidence Software Technologies (Peking University), MOE, China  
<sup>2</sup>Department of Computer Science, North Carolina State University, USA  
{wangxy06,zhanglu,xiongyf04,meih}@sei.pku.edu.cn, xie@csc.ncsu.edu

### ABSTRACT

Web applications are becoming increasingly popular nowadays. During the development and evolution of a web application, a typical type of tasks is to change the presentation of the web application, such as correcting display errors, adding user-interface controls, or changing appearance styles. To change the presentation of a static web page, developers are able to modify the HTML text of the web page using a graphical web-page editor. However, to change the presentation of a dynamic web application, developers need a graphical web-page editor to directly modify generated web pages, developers need to modify the code that generates the web pages.

### 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 [4, 3, 19], static checking for bugs in web applications [10, 33], and refactoring web applications [31, 18]. A typical type of daily tasks during the development and evolution of web applications is presentation changes, which are modifications made to change the appearance of web pages. Typical presentation changes in web applications include correction of display errors, adding user-interface controls, etc. According to our investigation of 600 bug reports from three

Wang et al. (2012)

The screenshot shows a web development tool interface. On the left, the 'Source' panel displays JavaScript code for a 'keyup' event handler and a 'render' function. The 'keyup' function updates a string 'str' with the value of the event target. The 'render' function returns an HTML snippet containing an input field with the value of 'str' and an 'onkeyup' event handler. On the right, the 'Time Control' panel features a 'Version' slider set to 1 and a 'State' slider set to 1. Below these are tabs for 'Output', 'HTML', and 'State'. The 'Output' tab is active, showing the text 'Weird, my keyboard just switched to Chinese.' with a small '2' next to it. The 'HTML' tab shows the rendered HTML snippet, and the 'State' tab shows the current state of the application.

Schuster & Flanagan (2016)

But we want to track any value of interest to output.

# Two Key Ideas

1. Generic dependency tracing.

# Transparent ML (TML) [Acar et al. 2013]

Specifically, the **dependency provenance** scheme.

## A Core Calculus for Provenance

Umut A. Acar

Amal Ahmed

James Cheney

Roly Perera

June 4, 2018

### 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,

299v2 [cs.PL] 3 Jan 2014

# 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" ↓ "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()<sup>{1.●}</sup>, Before(10<sup>{2.1.●}</sup>, True<sup>{2.2.●}</sup>)<sup>{2.●}</sup>)<sup>{●}</sup>

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

↓ Tag subvalues with identifiers.

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

↓ Call toString using TML.

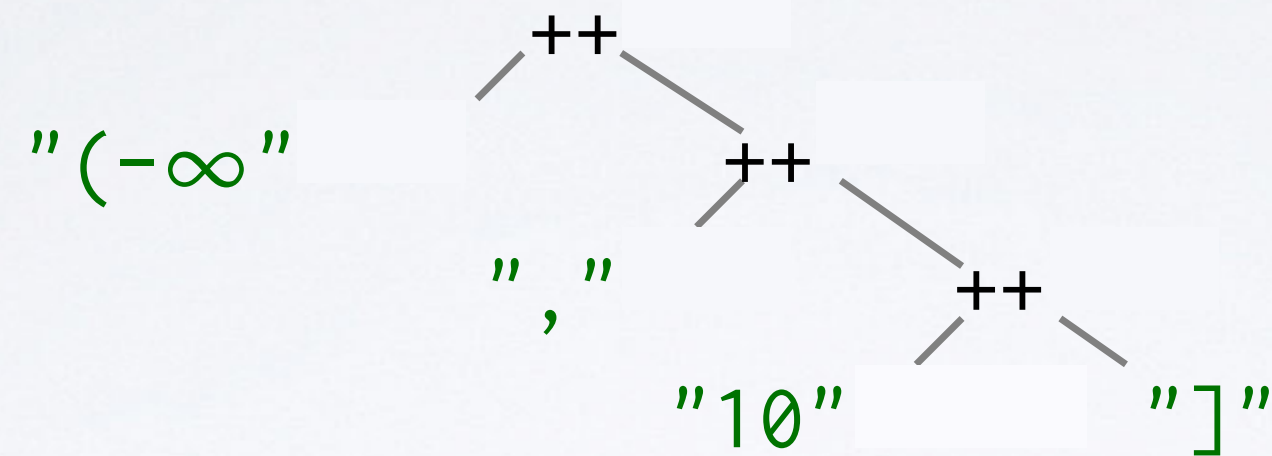
("(-∞" ++ ("," ++ ("10" ++ "]")))

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

↓ Tag subvalues with identifiers.

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

↓ Call toString using TML.

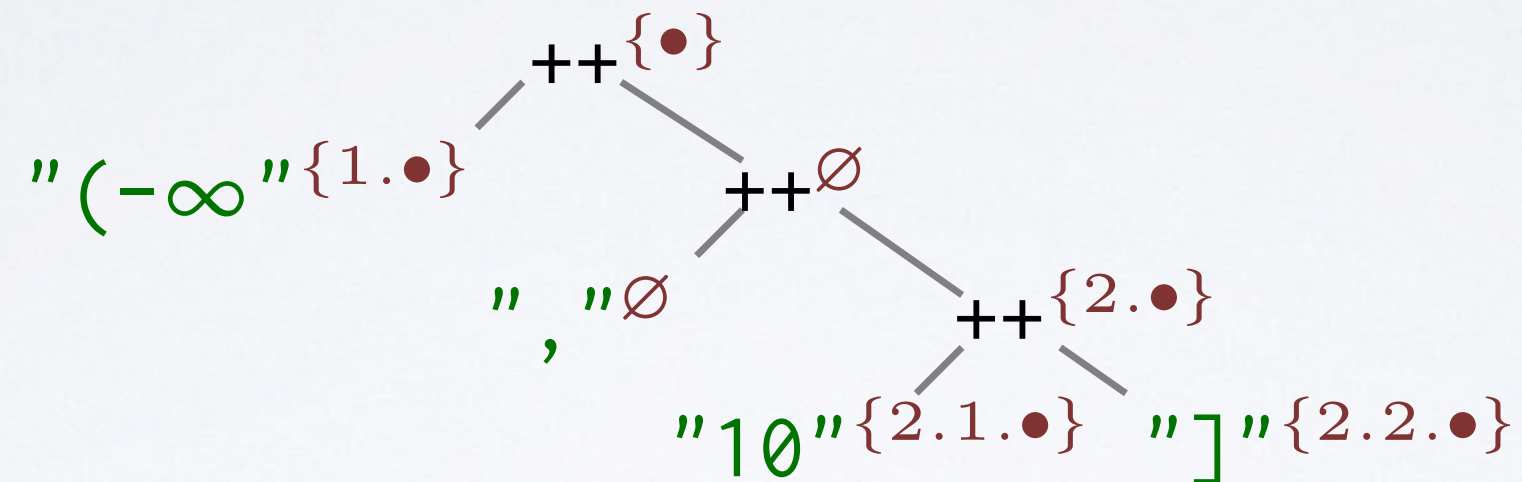


Interval(NegInf(), Before(10, True))

↓ Tag subvalues with identifiers.

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

↓ Call toString using TML.



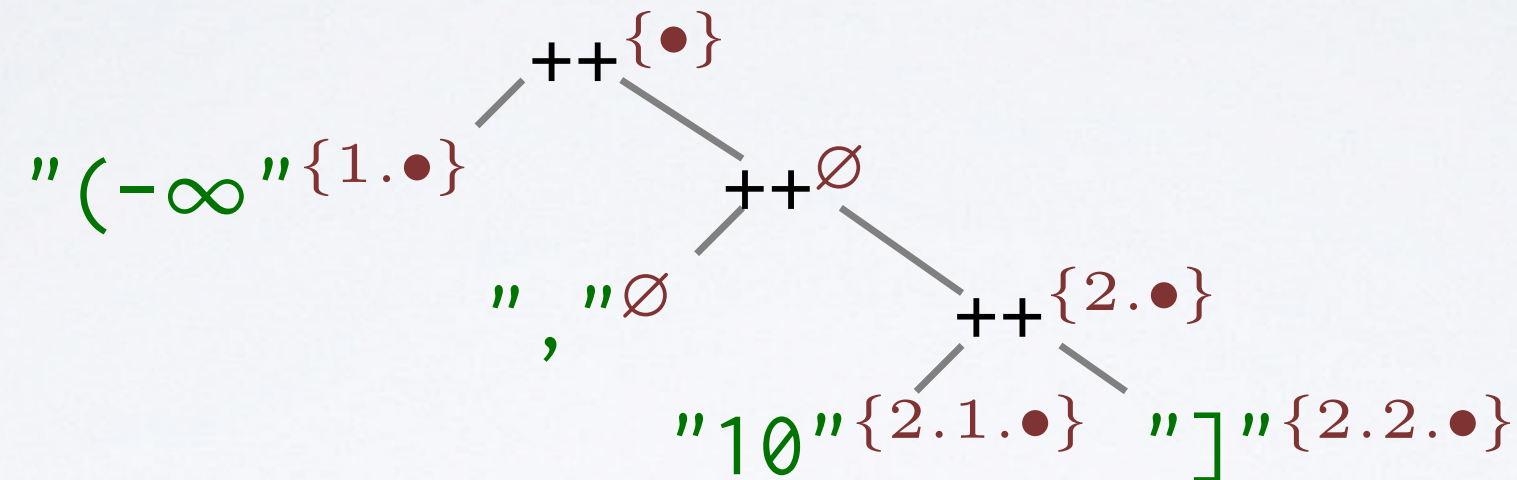


Interval(NegInf(), Before(10, True))

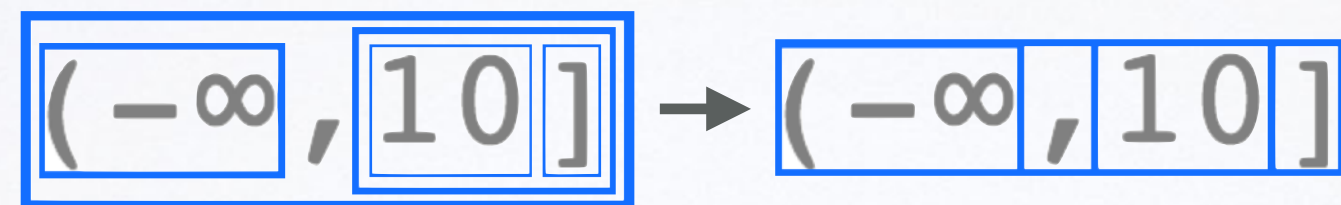
↓ Tag subvalues with identifiers.

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

↓ Call toString using TML.



↓ Assign spacial regions.

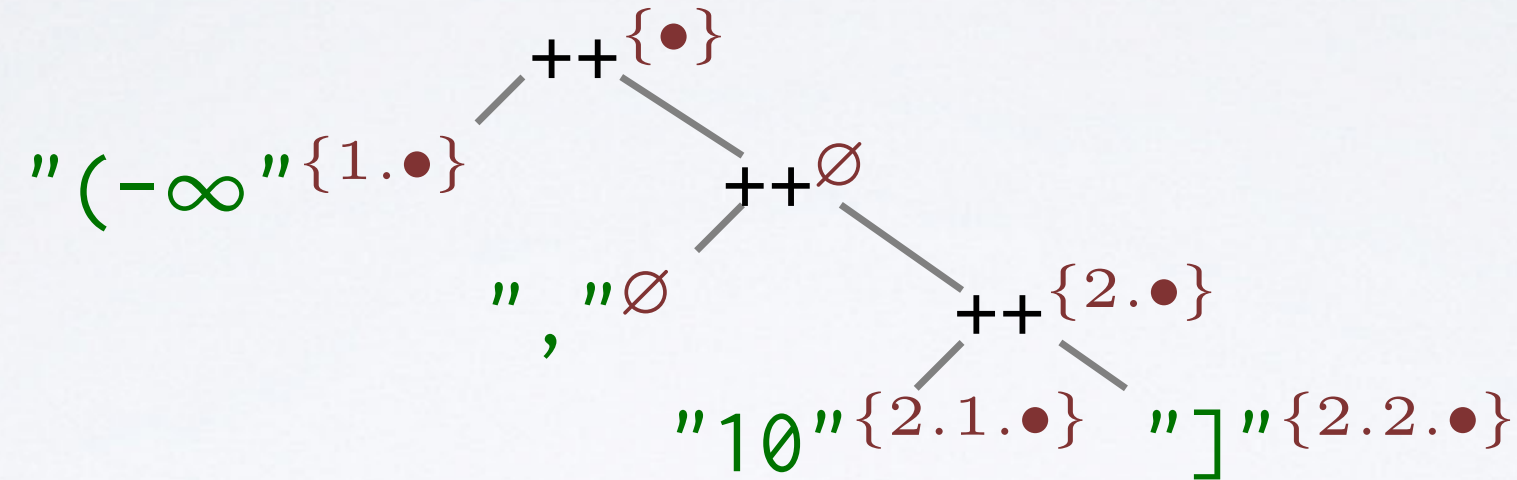


Interval(NegInf(), Before(10, True))

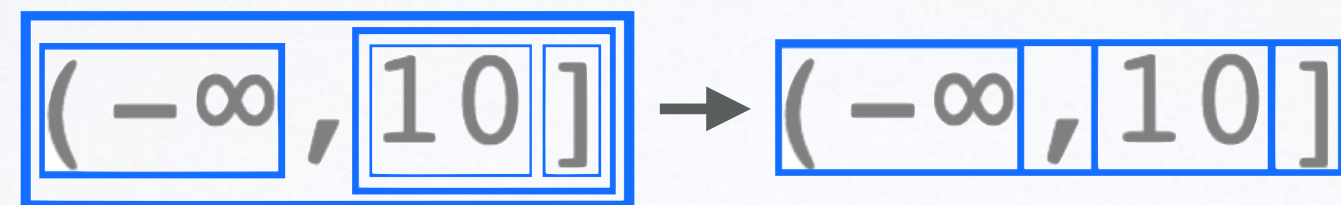
↓ Tag subvalues with identifiers.

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

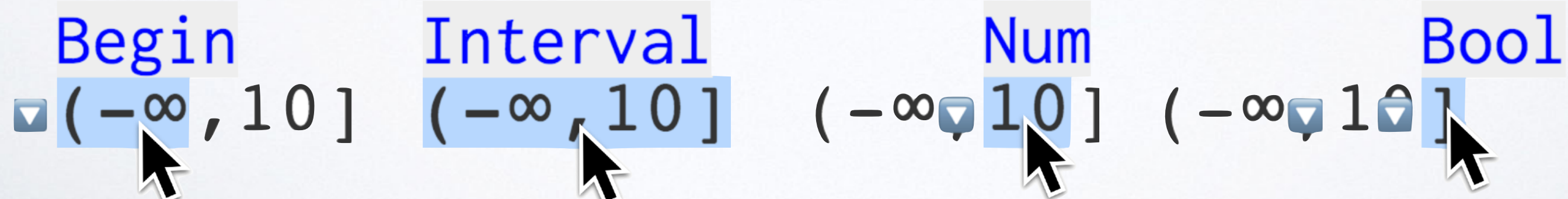
↓ Call toString using TML.



↓ Assign spacial regions.



↓ Assign action buttons (▾ swaps variants).



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

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

```
( $-\infty$ , 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]
```

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

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

```
[⊕1, ⊕2, ⊕3]
```

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

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

```
[⊕1, ⊕2, ⊕3]
```

# Case Studies

| Data Structure | %Selectable |
|----------------|-------------|
| Subvalues      | Items       |

---

# Case Studies

| Data Structure            | %Selectable |       |
|---------------------------|-------------|-------|
|                           | Subvalues   | Items |
| Interval                  |             |       |
| Date                      |             |       |
| JSON (multiline)          |             |       |
| List                      |             |       |
| List ("]" in base case)   |             |       |
| List (via join)           |             |       |
| List (via different join) |             |       |
| Tree (S-exp)              |             |       |
| Tree (indented hierarchy) |             |       |



# Case Studies

| Data Structure            | %Selectable |       |
|---------------------------|-------------|-------|
|                           | Subvalues   | Items |
| Interval                  |             |       |
| Date                      |             |       |
| JSON (multiline)          |             |       |
| List                      |             |       |
| List ("]" in base case)   |             |       |
| List (via join)           |             |       |
| List (via different join) |             |       |
| Tree (S-exp)              |             |       |
| Tree (indented hierarchy) |             |       |
| Pair [22]                 |             |       |
| List [22]                 |             |       |
| ADT (recursive) [22]      |             |       |
| Record [22]               |             |       |
| Set [13]                  |             |       |

# Case Studies

| Data Structure            | %Selectable |       |
|---------------------------|-------------|-------|
|                           | Subvalues   | Items |
| Interval                  | 80% (4/5)   |       |
| Date                      | 100% (4/4)  |       |
| JSON (multiline)          | 33% (14/43) |       |
| List                      | 86% (6/7)   |       |
| List ("]" in base case)   | 100% (7/7)  |       |
| List (via join)           | 71% (5/7)   |       |
| List (via different join) | 86% (6/7)   |       |
| Tree (S-exp)              | 53% (10/19) |       |
| Tree (indented hierarchy) | 21% (4/19)  |       |
| Pair [22]                 | 100% (3/3)  |       |
| List [22]                 | 100% (7/7)  |       |
| ADT (recursive) [22]      | 100% (4/4)  |       |
| Record [22]               | 100% (9/9)  |       |
| Set [13]                  |             |       |

# Case Studies

| Data Structure            | %Selectable |            |
|---------------------------|-------------|------------|
|                           | Subvalues   | Items      |
| Interval                  | 80% (4/5)   |            |
| Date                      | 100% (4/4)  |            |
| JSON (multiline)          | 33% (14/43) |            |
| List                      | 86% (6/7)   | 100% (3/3) |
| List ("]" in base case)   | 100% (7/7)  | 100% (3/3) |
| List (via join)           | 71% (5/7)   | 100% (3/3) |
| List (via different join) | 86% (6/7)   | 100% (3/3) |
| Tree (S-exp)              | 53% (10/19) | 100% (5/5) |
| Tree (indented hierarchy) | 21% (4/19)  | 100% (5/5) |
| Pair [22]                 | 100% (3/3)  |            |
| List [22]                 | 100% (7/7)  | 100% (3/3) |
| ADT (recursive) [22]      | 100% (4/4)  |            |
| Record [22]               | 100% (9/9)  |            |
| Set [13]                  |             | 100% (4/4) |

# TSE Limitations

# TSE Limitations


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

# TSE Limitations

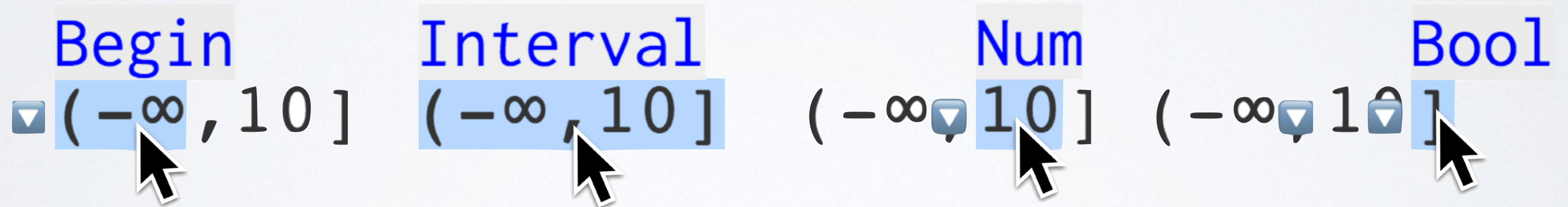
- Standalone prototype — needs to be adapted to some concrete setting.

- Occlusion. The diagram shows two overlapping rectangular boxes with blue borders. The left box contains the characters '10]' and the right box contains '10]'. The right box is positioned slightly lower and to the right of the left box, causing it to partially overlap the bottom-right corner of the left box. This visualizes the concept of occlusion where one element is partially hidden by another.

# TSE Limitations

- Standalone prototype — needs to be adapted to some concrete setting.
- Occlusion. 
- Insert/remove only works well for lists.

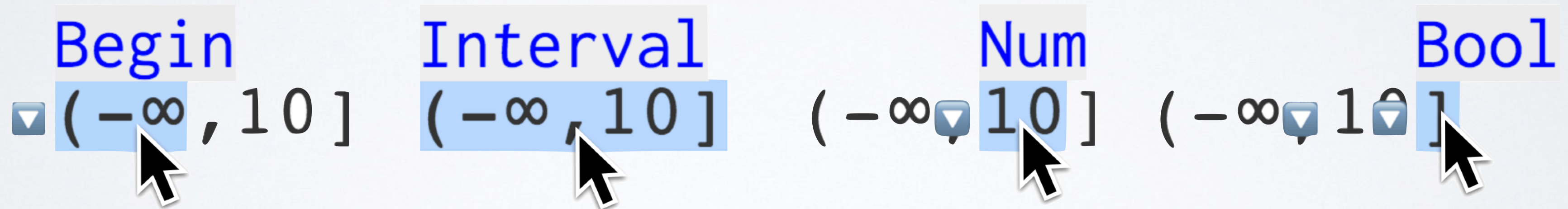
# Tiny Structure Editors for Low, Low Prices!





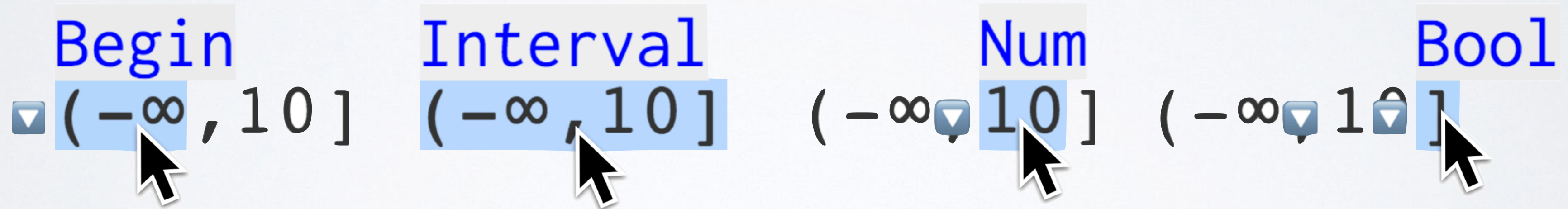
# Tiny Structure Editors for Low, Low Prices!

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



# Tiny Structure Editors for Low, Low Prices!

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

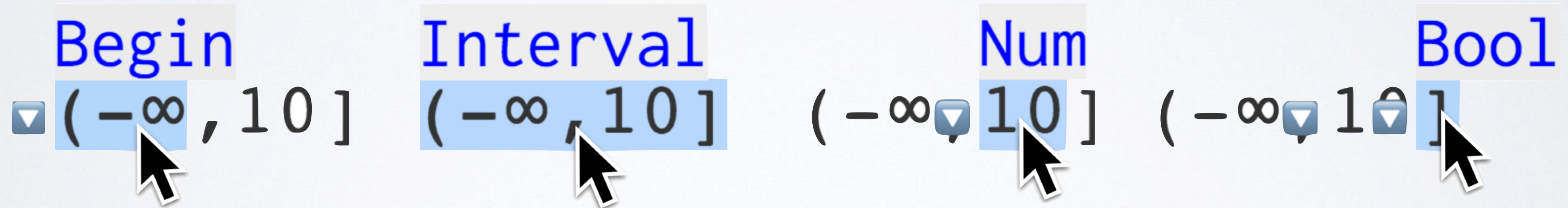


⇒ Custom, manipulable visualizations for user-defined types.

# Tiny Structure Editors for Low, Low

*Thank  
You!*

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



⇒ Custom, manipulable visualizations for user-defined types.



# TML in TSE

|                                  |       |  |
|----------------------------------|-------|--|
| <b>Expressions</b> $e$           | $::=$ | $x \mid f(x).e \mid e_1(e_2)$<br>$\mid C(e_1, \dots, e_n)$<br>$\mid \text{case } e \text{ of } \overline{C_i(x_1, \dots, x_n) \rightarrow e_i}$<br>$\mid s \mid e_1 ++ e_2 \mid \text{strLen}(e)$<br>$\mid n \mid e_1 \oplus e_2 \mid \text{numToStr}(e)$<br>$\mid \text{basedOn}(e_d, e)$ |
| <b>Projection Paths</b> $\pi$    | $::=$ | $\bullet \mid i.\pi$   |
| <b>(Tagged) Values</b> $w$       | $::=$ | $v\{\pi_1, \dots, \pi_n\}$   |
| <b>(Untagged) Pre-Values</b> $v$ | $::=$ | $[E] f(x).e \mid C(w_1, \dots, w_n)$<br>$\mid s \mid w_1 ++ w_2 \mid n$<br>$\mid \text{dynCall}(f)$  |
| <b>Tagged Environments</b> $E$   | $::=$ | $- \mid E, x \mapsto w$  |

## Evaluation with Dependency Provenance

$$\boxed{E \vdash e \Downarrow w}$$

[EVALCASE]

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

```
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))
```

**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))
```

**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))
```

$(-\infty, 10]$

```
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 ")")
```

$(-\infty, 10]$

```
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))
```

$(-\infty, 10]$



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

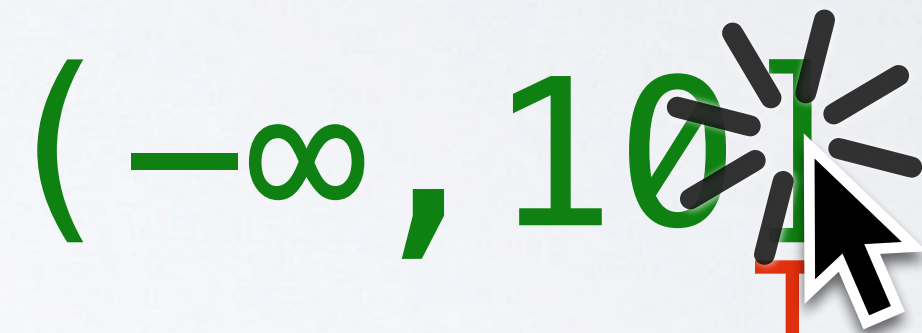
```
toString : Begin -> String
toString(begin) = case begin of
  NegInf()          -> "(-∞"
  After(num, isClosed) ->
    (if isClosed then "[" else "(") ++ toString(num)
```

```
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))
```

(-∞, 10]



```
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)
```

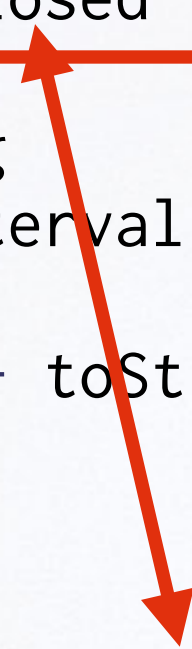
```
toString : End -> String
toString(end) = case end of
  Inf()          -> "∞)"
  Before(num, isClosed) ->
    toString(num) ++ (if isClosed then "]" else ")")
```

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

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

$(-\infty, 10)$

`(if isClosed then "]" else ")")`



```
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))
```

$(-\infty, 10]$



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

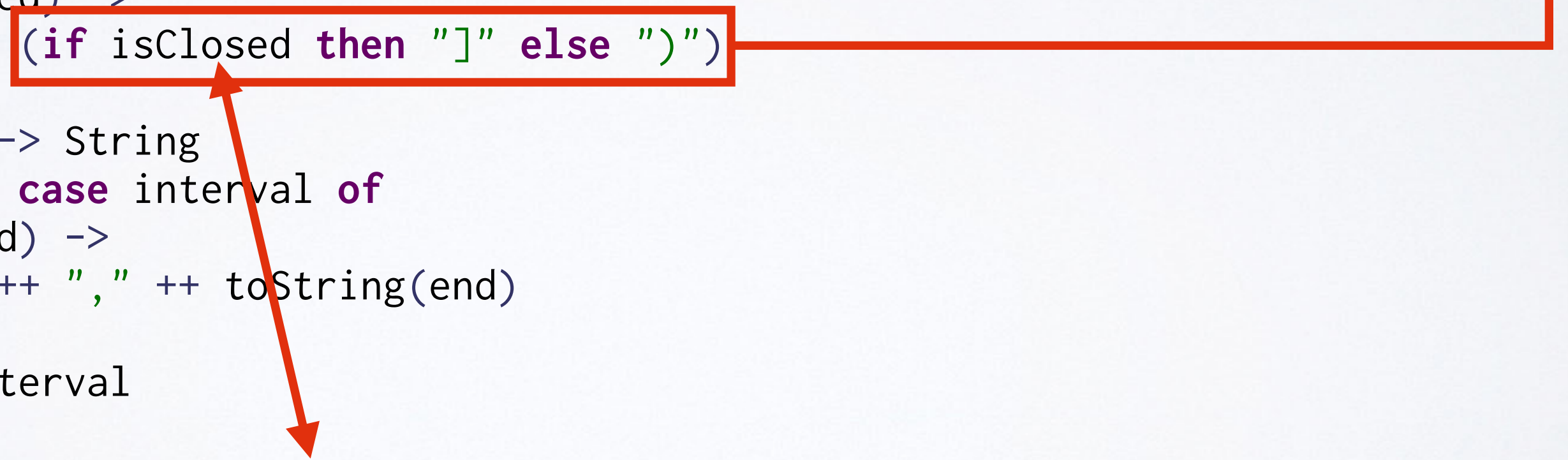
```
toString : Begin -> String
toString(begin) = case begin of
  NegInf()          -> "(-∞"
  After(num, isClosed) ->
    (if isClosed then "[" else "(") ++ toString(num)
```

```
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))
```

$(-\infty, 10)$



```
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))
```

$(-\infty, 10]$





```
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))
```

$(-\infty, 10)$



