Higher-Order Demand-Driven Symbolic Evaluation

PLUM Reading Group
Shiwei Weng, Sep 2020
Outline

● Motivation
● Demand-driven functional interpreter
● Demand-driven symbolic evaluator
● Implementation
● Current status
Motivation

- Generate tests to execute to the specified program points
Motivation

- Generate tests to execute to the specified program points
- Execute from the interested point backwards to the start point
Motivation

- Generate tests to execute to the specified program points
- Execute from the interested point backwards to the start point
- Benefit from the demand-driven technique
  - Goal-directed, backward-chain in logic programming, laziness, directed
  - Fewer spurious paths taken
Motivation

- Generate tests to execute to the specified program points

- Execute from the interested point backwards to the start point

- Benefit from the demand-driven technique
  - Goal-directed, backward-chain in logic programming, laziness, directed
  - Fewer spurious paths taken

- Continuing work of demand-driven program analysis
Outline

- Motivation
- **Demand-driven** functional interpreter
- **Demand-driven** symbolic evaluator
Demand-driven functional interpreter

- Start from the end
- No substitution, environments or closures
- Find the binding when needed
Demand-driven functional interpreter

- Start from the end (any top-level program point)
- No substitution, environments or closures
- Find Lookup the value of a variable, along (the graph of) source code
- Lookup is the interpreter
Demand-driven functional interpreter

```haskell
let y = 0 in
let f = (fun x ->
        let fret = x + 1 in
        fret) in
let fy = f y in
let f1 = f 1 in
let ret = fy + f1 in
ret
```

- Lookup, $\mathbb{L}([x], @x_{pp}, [...]) \equiv v$
  - $x$ is the variable to lookup
  - $x_{pp}$ is the program point to start the lookup
  - $[...]$ is the stack of call frames
Demand-driven functional interpreter

- \( \mathbb{L}([y], @y, []) \equiv 0 \)
- \( \mathbb{L}([y], @fy, []) \equiv 0 \)
- \( \mathbb{L}([fret], @fret, [fy]) \equiv 1 \)
- \( \mathbb{L}([f1], @f1, []) \equiv 2 \)

- Lookup, \( \mathbb{L}([x], @X_{pp}, [...]) \equiv v \)
  - \( x \) is the variable to lookup
  - \( X_{pp} \) is the program point to start the lookup
  - \( [...] \) is the stack of call frames

JavaScript code snippet:

```javascript
let y = 0 in
let f = (fun x ->
    let fret = x + 1 in
    fret
) in
let fy = f y in
let f1 = f 1 in
let ret = fy + f1 in
ret
```
Demand-driven functional interpreter

```
let y = 0 in
let f = (fun x →
    let fret = x + 1 in
    fret) in
let fy = f y in
let f1 = f 1 in
let ret = fy + f1 in
ret
```

- **Lookup**, $\mathcal{L}(\left[ x \right], \@_{x_{pp}}, [...]) \equiv v$
  - $x$ is the variable to lookup
  - $x_{pp}$ is the program point to start the lookup
  - $[...]$ is the stack of call frames

- $\mathcal{L}([fy], \@_{fy}, [])$
  $\equiv \mathcal{L}([fret], \@_{fret}, [fy])$
Demand-driven functional interpreter

```
let y = 0 in
let f = (fun x ->
    let fret = x + 1 in
    fret)
in
let fy = f y in
let f1 = f 1 in
let ret = fy + f1 in
ret
```

- Lookup, $\mathbb{L}([x], @x_{pp}, [...]) \equiv v$
  - $x$ is the variable to lookup
  - $x_{pp}$ is the program point to start the lookup
  - $[...]$ is the stack of call frames

- $\mathbb{L}([fy], @fy, [])$
  $\equiv \mathbb{L}([fret], @fret, [fy])$
  $\equiv \mathbb{L}([x], @fret, [fy]) + 1$
Demand-driven functional interpreter

\[
\begin{align*}
\text{let } & y = 0 \text{ in } \\
\text{let } & f = (\text{fun } x \rightarrow \\
& \quad \text{let } fret = x + 1 \text{ in } \\
& \quad \text{fret}) \text{ in } \\
\text{let } & fy = f \ y \text{ in } \\
\text{let } & f1 = f \ 1 \text{ in } \\
\text{let } & ret = fy + f1 \text{ in } \\
\text{ret}
\end{align*}
\]

- Lookup, \( \mathbb{L}([x], @x_{pp}, [...]) \equiv v \)
  - \( x \) is the variable to lookup
  - \( x_{pp} \) is the program point to start the lookup
  - \([...]\) is the stack of call frames

- \( \mathbb{L}([fy], @fy, []) \)
  \( \equiv \mathbb{L}([fret], @fret, [fy]) \)
  \( \equiv \mathbb{L}([x], @fret, [fy]) + 1 \)

- \( \mathbb{L}([x], @fret, [fy]) \)
  \( \equiv \mathbb{L}([x], @\text{fun } x->, [fy]) \equiv \mathbb{L}([y], @fy, []) \)
Demand-driven functional interpreter

\[
\begin{align*}
&\text{let } y = 0 \text{ in} \\
&\text{let } f = (\text{fun } x \rightarrow \text{let } fret = x + 1 \text{ in} \text{fret}) \text{ in} \\
&\text{let } fy = f \ y \text{ in} \\
&\text{let } f1 = f \ 1 \text{ in} \\
&\text{let } ret = fy + f1 \text{ in} \\
&\text{ret}
\end{align*}
\]

- **Lookup**, \(\mathbb{L}([x], @x_{pp}, [...]) \equiv v\)
  - \(x\) is the variable to lookup
  - \(x_{pp}\) is the program point to start the lookup
  - [... is the stack of call frames

- \(\mathbb{L}([fy], @fy, [])\)
  \(\equiv \mathbb{L}([fret], @fret, [fy])\)
  \(\equiv \mathbb{L}([x], @fret, [fy]) + 1\)

- \(\mathbb{L}([x], @fret, [fy])\)
  \(\equiv \mathbb{L}([x], @\text{fun } x->, [fy]) \equiv \mathbb{L}([y], @fy, [])\)
  \(\equiv \mathbb{L}([y], @f, []) \equiv \mathbb{L}([y], @y, []) \equiv 0\)

- \(\mathbb{L}([fy], @fy, [[]])\)
  \(\equiv \mathbb{L}([fret], @fret, [[fy]])\)
  \(\equiv \mathbb{L}([x], @fret, [[fy]]) + 1\)

- \(\mathbb{L}([x], @fret, [[fy]])\)
  \(\equiv \mathbb{L}([x], @\text{fun } x->, [[fy]]) \equiv \mathbb{L}([y], @fy, [[]])\)
  \(\equiv \mathbb{L}([y], @f, [[]]) \equiv \mathbb{L}([y], @y, [[]]) \equiv 0\)
Demand-driven functional interpreter

- **Lookup**, \( L([x], @X_{pp}, [...]) \equiv v \)
  - \( x \) is the variable to lookup
  - \( X_{pp} \) is the program point to start the lookup
  - \([...]\) is the stack of call frames

```plaintext
let y = 0 in
let f = (fun x ->
  let fret = x + 1 in
  fret
) in
let fy = f y in
let f1 = f 1 in
let ret = fy + f1 in
ret
```

- \( L([fy], @fy, []) \)
  \( \equiv L([fret], @fret, [fy]) \)
  \( \equiv L([x], @fret, [fy]) + 1 \)
  \( \equiv 0 + 1 \)
  \( \equiv 1 \)
Lookup a nonlocal variable

```
let g = (fun x ->
  let gret = (fun y ->
    let gyret = x + y in gyret)
  in gret) in
let g5 = g 5 in
let ret = g5 1 in ret
```

- Lookup, $L([x], @x_{pp}, [...]) \equiv v$
  - $x$ is the variable to lookup
  - $x_{pp}$ is the program point to start the lookup
  - $[...]$ is the stack of call frames

- Step 1: find the definition site for $g5$
  Step 2: resume search for $x$ since that is lexical scope of its definition
Lookup a nonlocal variable

```
let g = (fun x ->
    let gret = (fun y ->
        let gyret = x + y in gyret)
    in gret) in
let g5 = g 5 in
let ret = g5 1 in ret
```

- Lookup, $\mathbb{L}([\text{xs}], @\text{pp}, [...]) \equiv v$
  - $\text{xs}$ is the sequence of variable to lookup
  - $\text{pp}$ is the program point to start the lookup
  - $[...]$ is the stack of call frames

- Step 1: find the definition site for $g5$
Lookup a nonlocal variable

- \( \mathbb{L}(\llbracket x \rrbracket, \mathit{@gyret}, \mathbb{L}) \equiv 5 \)
- Step 2: resume search for \( x \) since that is lexical scope of its definition

```ocaml
let g = (fun x ->
    let gret = (fun y ->
        let gyret = x + y in gyret
    ) in gret)
let g5 = g 5 in
let ret = g5 1 in ret
```

- Lookup, \( \mathbb{L}(\llbracket xs \rrbracket, \mathit{@xpp}, [...] \) \) \( \equiv v \)
  - \( xs \) is the sequence of variable to lookup
  - \( xpp \) is the program point to start the lookup
  - \( [...] \) is the stack of call frames
Demand-driven functional interpreter

- Lookup, \( \mathbb{L}([X_{f1}, X_{f2}, \ldots, X], @X_{pp}, [...]) = v \)
- No substitution, environments or closures
- Start from
  - the end or any toplevel program point, when we know the [...] is empty
  - Any program point, if we know the call stack
Demand-driven functional interpreter

- Lookup, $\mathbb{L}(\langle x_1, x_2, \ldots, x \rangle, @x_{pp}, \ldots) \equiv v$
- No substitution, environments or closures
- Start from
  - the end or any toplevel program point, when we know the $\ldots$ is empty
  - Any program point, if we know the call stack
- Support input `let x = input in ...`
- Support records and recursive data structures
- Recursion encoded via self-passing (currently)
- Implemented in ANF with unique variable names
Lookup rules

\[
\begin{align*}
\text{Value Discovery} & \quad \frac{\text{FIRST}(x, \text{CL}(x), C)}{\mathcal{L}([[x]], (x = \nu), C) \equiv \nu} \\
\text{Value Discard} & \quad \frac{\mathcal{L}(X, \text{PRED}(x), C) \equiv \nu}{\mathcal{L}([[x]], X, (x = f), C) \equiv \nu} \\
\text{Alias} & \quad \frac{\mathcal{L}([x'], X, \text{PRED}(x), C) \equiv \nu}{\mathcal{L}([[x]], X, (x = x'), C) \equiv \nu} \\
\text{Function Enter} & \quad \frac{c = (x_r = x_f x_v)}{\mathcal{L}([x_v], X, \text{PRED}(c), C) \equiv \nu} \\
\text{Function Exit} & \quad \frac{\text{RET}_\text{CL}(e) = (x' = b)}{\mathcal{L}([x_f], \text{PRED}(c), C) \equiv \nu} \\
\text{Parameter} & \quad \frac{x'' \neq x}{\mathcal{L}([x], (\text{fun } x \to), [c] || C) \equiv \nu} \\
\text{Non-Local} & \quad \frac{c = (x_r = x_f x_v)}{\mathcal{L}([x_f], X, \text{PRED}(c), C) \equiv \nu} \\
& \quad \frac{\mathcal{L}([x'], X, (x' = b), [\text{CL}(x)] || C) \equiv \nu}{\mathcal{L}([x_f], \text{PRED}(c), C) \equiv [\text{fun } x'' \to] || e} \\
& \quad \frac{\mathcal{L}([x], (\text{fun } x'' \to), [c] || C) \equiv \nu}{\mathcal{L}([x_f], \text{PRED}(c), C) \equiv [\text{fun } x'' \to] || e} \\
\text{Skip} & \quad \frac{x'' \neq x}{\mathcal{L}([x], X, \text{PRED}(x''), C) \equiv \nu} \\
& \quad \frac{\exists \nu_0. \mathcal{L}([x'''], \text{CL}(x''), C) \equiv \nu_0}{\mathcal{L}([x], X, (x'' = b), C) \equiv \nu} 
\end{align*}
\]
From concrete to symbolic

- $\mathbb{L}([x_{f1}, x_{f2}, \ldots, x], @x_{pp}, [...]) \equiv \mathbb{L}(\ldots) \equiv \mathbb{L}(\ldots) \equiv v$

- $\mathbb{L}^s([x_{f1}, x_{f2}, \ldots, x], @x_{pp}, [...]), \mathbb{L}(\ldots), \mathbb{L}(\ldots) \equiv s v \text{ over } \Phi$
From concrete to symbolic

- $\mathbb{L}([x_{f1}, x_{f2}, \ldots, X], @X_{pp}, [...] ) \equiv \mathbb{L}(...) \equiv \mathbb{L}(...) \equiv v$
  - Deterministic $v$ (Lemma 3.4, at most one $v$ s.t a proof can be constructed)
  - A reverse interpreter is sound and complete with respect to a forward one
    - Need to know the call stack
    - Need to sort the input order

- $\mathbb{L}^s([x_{f1}, x_{f2}, \ldots, X], @X_{pp}, [...] )$, $\mathbb{L}(...) , \mathbb{L}(...) \equiv^s v$ over $\Phi$
  - Nondeterministic
    - Not know the call stack
    - Not know the input
  - $\Phi$ equationally constraints variables, must be satisfiable
Functional symbolic interpreter

```
let y = 0 in
let f = (fun x ->
    let fret = x + 1 in
    fret) in
let fy = f y in
let f1 = f 1 in
let ret = fy + f1 in
```

- \( \mathbb{L}([fy], @fy, []) \)
  \( \equiv \mathbb{L}([fret], @fret, [fy]) \)
  \( \equiv \mathbb{L}([x], @fret, [fy]) + 1 \)
- \( \mathbb{L}([x], @fret, [fy]) \)
  \( \equiv \mathbb{L}([x], @fun x->, [fy]) \)
  \( \equiv \mathbb{L}([y], @fy, []) \)
Functional symbolic interpreter

```ocaml
let y = 0 in
let f = (fun x ->
    let fret = x + 1 in
    fret) in
let fy = f y in
let f1 = f 1 in
let ret = fy + f1 in

- $\mathbb{L}_s([fy], \odot fy, [])$
  $\equiv \mathbb{L}_s([fret], \odot fret, [fy]), \Phi^1$
  $\equiv \mathbb{L}_s([x], \odot fret, [fy]) + 1, \Phi^2$
- $\mathbb{L}_s([x], \odot fret, [fy])$
  $\equiv \mathbb{L}_s([x], \odot \text{fun } x->, [fy]), \Phi^3$
  $\equiv \mathbb{L}_s([y], \odot fy, []), \Phi^4$
```
Functional symbolic interpreter

```
let y = 0 in
let f = (fun x ->
    let fret = x + 1 in
    fret) in
let fy = f y in
let f1 = f 1 in
let ret = fy + f1 in
ret
```

- \( \mathbb{L}^s([fy], @fy, []) \)
  \[ \equiv \mathbb{L}^s([fret], @fret, [fy]), \Phi^1 = \{ \; ^\uparrow f y = [fy]fret \} \]
  \[ \equiv \mathbb{L}^s([x], @fret, [fy]) + 1, \Phi^2 = \{ \; ^\uparrow f y = [fy]x + 1 \} \]
- \( \mathbb{L}^s([x], @fret, [fy]) \)
  \[ \equiv \mathbb{L}^s([x], @fun x->, [fy]), \Phi^3 = \{ \; \} \]
  \[ \equiv \mathbb{L}^s([y], @fy, []), \Phi^4 = \{ \; ^\uparrow f y = [fy]x = ^\downarrow y \} \]
Functional symbolic interpreter

```
let y = 0 in
let f = (fun x ->
    let fret = x + 1 in
    fret)
in
let fy = f y in
let f1 = f 1 in
let ret = fy + f1 in
ret
```

- $\mathbb{L}^s([fy], @fy, [])$
  \[ \equiv \mathbb{L}^s([fret], @fret, [fy]), \Phi^1 = \{ \text{[]fy = [fy]fret} \} \]
  \[ \equiv \mathbb{L}^s([x], @fret, [fy]) + 1, \Phi^2 = \{ \text{... , [fy]ret = [fy]x + 1} \} \]

- $\mathbb{L}^s([x], @fret, [fy])$
  \[ \equiv \mathbb{L}^s([x], @\text{fun x->}, [fy]), \Phi^3 = \{ \text{... } \} \]
  \[ \equiv \mathbb{L}^s([y], @fy, []), \Phi^4 = \{ \text{... , [fy]x = []y} \} \]

satisfiable, not interesting
Relative stack

\[
\text{let } y = 0 \text{ in }
\text{let } f = (\text{fun } x \rightarrow \\
\hspace{1cm} \text{let } \text{fret} = x + 1 \text{ in } \\
\hspace{2cm} \text{fret}) \text{ in }
\text{let } \text{fy} = f \ y \text{ in } 
\text{let } \text{f1} = f \ 1 \text{ in } 
\text{let } \text{ret} = \text{fy} + \text{f1} \text{ in } 
\text{ret}
\]

- \( \mathbb{L}^s([\text{fy}], \text{fret}, []) \)
  \( \mathbb{L}^s([\text{fret}], \text{fret}, [\text{fy}]) \), \( \Phi_1 = \{ \text{fy} = [\text{fy}] \text{fret} \} \)
  \( \equiv \mathbb{L}^s([x], \text{fret}, [\text{fy}]) + 1, \Phi_2 = \{ \ldots, [\text{fy}] \text{ret} = [\text{fy}]x + 1 \} \)

- \( \mathbb{L}^s([x], \text{fret}, [\text{fy}]) \)
  \( \equiv \mathbb{L}^s([x], \text{fun } x \rightarrow, [\text{fy}]), \Phi_3 = \{ \ldots \} \)
  \( \equiv \mathbb{L}^s([y], \text{fy}, []) \), \( \Phi_4 = \{ \ldots, [\text{fy}]x = \emptyset \} \) satisfiable, not interesting
Relative stack

```plaintext
let y = 0 in
let f = (fun x ->
  let fret = x + 1 in
  fret) in
let fy = f y in
let f1 = f 1 in
let ret = fy + f1 in
ret
```

- $\mathbb{L}^s([fy], @fy, [])$
  $\mathbb{L}^s([fret], @fret, [])$
  $\equiv \mathbb{L}^s([x], @fret, []) + 1$, $\Phi^2 = \{ \ldots, \_\text{ret} = \_x + 1 \}$
- $\mathbb{L}^s([x], @fret, [])$
  $\equiv \mathbb{L}^s([x], @\text{fun } x->, []), \Phi^3 = \{ \ldots \}$
  $\equiv \mathbb{L}^s([y], @fy, [??]), \Phi^4 = \{ \ldots, \_x = [??]y \}$
Relative stack

```
let y = 0 in
let f = (fun x ->
  let fret = x + 1 in
  fret)
let fy = f y in
let f1 = f 1 in
let ret = fy + f1
ret
```

- $L^s([fy], @fy, [])$
  $L^s([fret], @fret, [])$
  $\equiv L^s([x], @fret, []) + 1, \Phi^2 = \{ \ldots, \Box \text{ret} = \Box x + 1 \}$

- $L^s([x], @fret, [])$
  $\equiv L^s([x], @fun x->, []), \Phi^3 = \{ \ldots \}$
  $\equiv L^s([y], @fy, [-fy]), \Phi^4 = \{ \ldots, \Box x = [-fy]y \}$
Relative stack

\[
\begin{align*}
\text{let } y &= 0 \text{ in} \\
\text{let } f &= (\text{fun } x \rightarrow \\
&\quad \text{let } \text{fret} = x + 1 \text{ in} \\
&\quad \text{fret}) \text{ in} \\
\text{let } \text{fy} &= f \text{ y in} \\
\text{let } f1 &= f 1 \text{ in} \\
\text{let } \text{ret} &= \text{fy} + f1 \text{ in}
\end{align*}
\]

\[
\begin{align*}
\mathbb{L}^s(\text{fret}, @\text{fret}, []) \\
\equiv \mathbb{L}^s([x], @\text{fret}, []) + 1, \Phi^2 = \{ ... , \neg \text{ret} = \neg x + 1 \} \\
\mathbb{L}^s([x], @\text{fret}, []) \\
\equiv \mathbb{L}^s([x], @\text{fun } x\rightarrow, []), \Phi^3 = \{ ... \} \\
\equiv \mathbb{L}^s([y], @\text{fy}, [-\text{fy}]), \Phi^4 = \{ ... , \neg x = [-f1]y \}
\end{align*}
\]
## Comparison of stacks

<table>
<thead>
<tr>
<th>Stack type</th>
<th>Concrete stack</th>
<th>Concrete stack</th>
<th>Relative stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter</td>
<td>Concrete</td>
<td>Concrete</td>
<td>Symbolic</td>
</tr>
<tr>
<td>Direction</td>
<td>Forward</td>
<td>Backward</td>
<td>Backward</td>
</tr>
<tr>
<td>Arrow</td>
<td>main -&gt; target</td>
<td>main &lt;- target</td>
<td>main &lt;- target</td>
</tr>
<tr>
<td>Value at main entry</td>
<td>[ ]</td>
<td>[ ]</td>
<td></td>
</tr>
<tr>
<td>Value at target</td>
<td>Can be non-empty</td>
<td>Can be non-empty</td>
<td></td>
</tr>
</tbody>
</table>
## Comparison of stacks

<table>
<thead>
<tr>
<th>Stack type</th>
<th>Concrete stack</th>
<th>Concrete stack</th>
<th>Relative stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter</td>
<td>Concrete</td>
<td>Concrete</td>
<td>Symbolic</td>
</tr>
<tr>
<td>Direction</td>
<td>Forward</td>
<td>Backward</td>
<td>Backward</td>
</tr>
<tr>
<td>Arrow</td>
<td>main -&gt; target</td>
<td>main &lt;- target</td>
<td>main &lt;- target</td>
</tr>
<tr>
<td>Value at main entry</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ -f1 ]</td>
</tr>
<tr>
<td>Value at target</td>
<td>Can be non-empty</td>
<td>Can be non-empty</td>
<td>[ ]</td>
</tr>
</tbody>
</table>
## Comparison of stacks

<table>
<thead>
<tr>
<th>Stack type</th>
<th>Concrete stack</th>
<th>Concrete stack</th>
<th>Relative stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter</td>
<td>Concrete</td>
<td>Concrete</td>
<td>Symbolic</td>
</tr>
<tr>
<td>Direction</td>
<td>Forward</td>
<td>Backward</td>
<td>Backward</td>
</tr>
<tr>
<td>Arrow</td>
<td><code>main -&gt; target</code></td>
<td><code>main &lt;- target</code></td>
<td><code>main &lt;- target</code></td>
</tr>
<tr>
<td>Value at <code>main</code> entry</td>
<td><code>[]</code> as empty</td>
<td><code>[]</code> as empty</td>
<td><code>[-f1]??[]</code></td>
</tr>
<tr>
<td>Value at <code>target</code></td>
<td>Can be non-empty</td>
<td>Can be non-empty</td>
<td><code>[]</code> as unknown</td>
</tr>
</tbody>
</table>
## Comparison of stacks

<table>
<thead>
<tr>
<th>Stack type</th>
<th>Concrete stack</th>
<th>Concrete stack</th>
<th>Relative stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter</td>
<td>Concrete</td>
<td>Concrete</td>
<td>Symbolic</td>
</tr>
<tr>
<td>Direction</td>
<td>Forward</td>
<td>Backward</td>
<td>Backward</td>
</tr>
<tr>
<td>Arrow</td>
<td>main -&gt; target</td>
<td>main &lt;- target</td>
<td>main &lt;- target</td>
</tr>
<tr>
<td>Value at main entry</td>
<td>[ ] as empty</td>
<td>[ ] as empty</td>
<td>[-f1]?[ ]</td>
</tr>
<tr>
<td>Value at target</td>
<td>Can be non-empty</td>
<td>Can be non-empty</td>
<td>[ ]?[ ]</td>
</tr>
</tbody>
</table>
Relative stack

**Definition 4.1.** Notation for pushing, popping, and concretizing relative stacks is as follows.

1. \( \text{Push}([c_1, \ldots, c_n]?[c_1', \ldots, c_n'], c) = [c_1, \ldots, c_n]?[c, c_1', \ldots, c_n'] \),
2. \( \text{Pop}([c_1, \ldots, c_n]?[], c) = [c, c_1, \ldots, c_n]?[] \),
3. \( \text{Pop}([c_1, \ldots, c_n]?[c_1', \ldots, c_n'], c) = [c_1, \ldots, c_n]?[c_2', \ldots, c_n'] \) for \( c = c_1' \),
4. \([c_1, \ldots, c_n]?[c_1', \ldots, c_n']\) is empty iff \( n' = 0 \) (the stack is empty, the co-stack may not be).
5. \( \text{Concretize}(C?[[]]) = \text{Reverse}(C) \)

- Push to the normal stack // rule (1)
- Pop from the empty normal stack, put into the co-stack // rule (2)
- Pop from the non-empty stack, it needs call-return alignment // rule (3)
- Safeguard: the normal stack must be empty when reaching the start // rule (4)
From concrete to symbolic

- \( \mathbb{L}(\{x_{f1}, x_{f2}, \ldots, x\}, @x_{pp}, [...]) \equiv v \)
  - Deterministic \( v \) (Lemma 3.4, at most one \( v \) s.t a proof can be constructed)
  - A reverse interpreter is sound and complete with respect to a forward one
    - Need to know the call stack
    - Need to sort the input order

- \( \mathbb{L}^s(\{x_{f1}, x_{f2}, \ldots, x\}, @x_{pp}, [...]) \), \( \mathbb{L}(...) \), \( \mathbb{L}(...) \equiv^s v \) over \( \Phi \)
  - Nondeterministic
    - Not know the call stack
    - Not know the input
  - \( \Phi \) equationally constraints variables, must be satisfiable
From concrete to symbolic, formally

- $\mathbb{L}^s(\mathcal{X}, \Phi, \Pi, c, \hat{\mathcal{C}}) \equiv^s s\nu$
  - $\mathcal{X}$ := lookup stack
  - $\Phi$ := constraint formulae
  - $\Pi$ := search path
  - $c$ := program point
  - $\hat{\mathcal{C}}$ := relative stack

Fig. 7. New Constructs for Symbolic Lookup
From concrete to symbolic, formally

- $\mathcal{L}^s(\mathcal{X}, \Phi, \Pi, c, \hat{C}) \equiv s_v$
  - $\mathcal{X} :=$ lookup stack
  - $\Phi :=$ constraint formulae
  - $\Pi :=$ search path
  - $c ::= $ program point
  - $\hat{C} ::= $ relative stack

- Checking along the lookup
  - $\Phi$ is satisfiable
  - $\Pi$ matches, a variable always points the same function in a nondeterministic trace
  - $\hat{C}$ has an empty normal stack part
Lookup rules, symbolically

\[
\begin{align*}
\dot{C}' &= \text{Pop}(\dot{C}, c) \\
\mathcal{L}^s([x_v] \| X, \text{pred}(c), \dot{C}') &\equiv \mathcal{C}_0 x_0
\end{align*}
\]

**Function Enter**

\[
\begin{align*}
c &= (x_f = x_f \ x_v) \\
\Pi(\dot{C}) &= c \\
\mathcal{L}^s([x_v] \| X, (\text{fun } x' \to (e)), \dot{C}) &\equiv \mathcal{C}_0 x_0
\end{align*}
\]

**Parameter**

\[
\begin{align*}
\dot{C}' &= \text{Pop}(\dot{C}, c) \\
X'' \neq x \\
c &= (x_r = x_f \ x_v) \\
\Pi(\dot{C}) &= c
\end{align*}
\]

**Function Enter**

\[
\begin{align*}
\mathcal{L}^s([x_f, x] \| X, \text{pred}(c), \dot{C}') &\equiv \mathcal{C}_0 x_0 \\
\mathcal{L}^s([x_f] \| X, \text{pred}(c), \dot{C}') &\equiv (\text{fun } x'' \to (e))
\end{align*}
\]

**Non-Local**

\[
\begin{align*}
\mathcal{L}^s([x] \| X, (\text{fun } x'' \to), \dot{C}) &\equiv \mathcal{C}_0 x_0
\end{align*}
\]

**Function Exit**

\[
\begin{align*}
\mathcal{L}^s([x'] \| X, (x' = b), \text{push}(\dot{C}, \text{cl}(x))) &\equiv \mathcal{C}_0 x_0 \\
\text{retCl}(e) &= (x' = b) \\
\mathcal{L}^s([x_f] \| X, \text{pred}(x), \dot{C}) &\equiv (\text{fun } x'' \to (e))
\end{align*}
\]

**Function Exit**

\[
\begin{align*}
x'' \neq x \\
\mathcal{L}^s([x] \| X, \text{pred}(x''), \dot{C}) &\equiv \mathcal{C}_0 x_0 \\
\mathcal{L}^s([x''] \| X, \text{cl}(x''), \dot{C}) &\equiv -
\end{align*}
\]

**Skip**

\[
\begin{align*}
x'' \neq x \\
\mathcal{L}^s([x] \| X, \text{pred}(x''), \dot{C}) &\equiv \mathcal{C}_0 x_0 \\
\mathcal{L}^s([x''] \| X, (x'' = b), \dot{C}) &\equiv \mathcal{C}_0 x_0
\end{align*}
\]
Lookup rules, symbolically

\[
\begin{align*}
\text{Value Discovery} & \quad \frac{x \neq \text{FIRSTV}(e_{\text{glob}}) \lor (\text{stack} = \text{CONCRETIZE}(\mathcal{C})) \in \Phi}{\mathcal{L}^S([x], (x = v), \mathcal{C}) \equiv \mathcal{C}_x} \\
\text{Input} & \quad \frac{\mathcal{C}_{\text{true}} = (\mathcal{C}_x = \mathcal{C}_x) \in \Phi}{\mathcal{L}^S([x], (x = \text{input}), \mathcal{C}) \equiv \mathcal{C}_x} \\
\text{Conditional Top} & \quad \frac{\mathcal{L}^S([x_2], \text{PRED}(x_1), \mathcal{C}) \equiv \beta}{\mathcal{L}^S(X, \text{PRED}(x_1), \mathcal{C}) \equiv \mathcal{C}_0 x_0} \quad \frac{\mathcal{L}^S([x_2], \text{PRED}(x_1), \mathcal{C}) \equiv \beta}{\mathcal{L}^S([x_2], \text{PRED}(x_1), \mathcal{C}) \equiv \beta} \\
\text{Conditional Bottom} & \quad \frac{\mathcal{L}^S([x'], || X, (x' = b), \mathcal{C}) \equiv \mathcal{C}_0 x_0}{\mathcal{L}^S([x_1], || X, (x_1 = x_2 ? e_{\text{true}} : e_{\text{false}}, \mathcal{C}) \equiv \mathcal{C}_0 x_0} \quad \text{RETCL}(e_{\beta}) = (x' = b)}
\end{align*}
\]

Fig. 8. Symbolic Lookup Rules
Outline

- Motivation
- Demand-driven functional interpreter
- Demand-driven symbolic evaluator
- Implementation
Implementation

- Artifact is a test generator: given program and target line, search for inputs which reach the target line of code
- Initial proof-of-concept implementation in OCaml
- Benchmark from Scheme Larceny and P4F
  - Modify by adding input
- Benchmark from Satisfiability Modulo Bounded Checking, CADE ’17
  - Add function to behave like uninterpreted one
Implementation

- Artifact is a test generator: given program and target line, search for inputs which reach the target line of code
- Initial proof-of-concept implementation in OCaml
- Benchmark from Scheme Larceny and P4F Thank you David!
  - Modify by adding input
- Benchmark from Satisfiability Modulo Bounded Checking, CADE ’17
  - Add function to behave like uninterpreted one
- Benchmark from Directed symbolic execution Thank you Mike!
  - Fully rewrite
  - Used in submissions before ICFP
Related Work

- Snugglebug, PLDI ’09
  Imperative demand symbolic execution, no correctness
- Satisfiability Modulo Bounded Checking, CADE ’17:
  Functional forward symbolic execution, no correctness proof, no input, no unbounded recursion
- Rosette, PLDI ’14
  a forward symbolic execution DSL; bounded datatypes only
- Our DDSE
  This work: functional, demand, arbitrary datatypes and recursion, proven
Current status

- Adapting on JavaScript
- Optimization
  - Mutable states
  - Cached lookup, formally
  - Function summarization
  - Pick among the spectrum between pure computational and pure SMT solving
Questions & suggestions