JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

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

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

```
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
```

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

- Lookup, $\mathbb{L}$([x], @x_{pp}, [...]) ≡ 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

```
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
```

- $\mathbb{L}$([fy], @fy, [])
  ≡ $\mathbb{L}$([fret], @fret, [fy])

- Lookup, $\mathbb{L}$([x], @x_{pp}, [...]) ≡ 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

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

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

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

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}, [...]) ≡ 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, [])
  ≡ $\mathbb{L}$([fret], @fret, [fy])
  ≡ $\mathbb{L}$([x], @fret, [fy]) + 1
- $\mathbb{L}$([x], @fret, [fy])
  ≡ $\mathbb{L}$([x], @fun x->, [fy]) ≡ $\mathbb{L}$([y], @fy, [])
  ≡ $\mathbb{L}$([y], @f, []) ≡ $\mathbb{L}$([y], @y, []) ≡ 0

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

- $\mathbb{L}([fy], @fy, [])$
  $\equiv \mathbb{L}([fret], @fret, [fy])$
  $\equiv \mathbb{L}([x], @fret, [fy]) + 1$
  $\equiv 0 + 1$
  $\equiv 1$

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

- $\mathbb{L}([x], @gyret, [ret])$
  $\equiv \mathbb{L}([g5, x], @ret, [])$
  $\equiv \mathbb{L}([gret, x], @gret, [g5])$

- Step 1: find the definition site for g5

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

- $\mathbb{L}([x], @gyret, [ret])$
  $\equiv \mathbb{L}([x], @fun\ x, [g5])$
  $\equiv \mathbb{L}([5], @g5, [])$
  $\equiv 5$

- Step 2: resume search for x since that is lexical scope of its definition

Demand-driven functional interpreter

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

$$\textsc{Value Discovery}\ \frac{\textsc{First}(x, \textsc{Cl}(x), C)}{\mathbb{L}([x], (x = v), C) \equiv v} \qquad \textsc{Value Discard}\ \frac{\mathbb{L}(X, \textsc{Pred}(x), C) \equiv v}{\mathbb{L}([x] \mid\mid X, (x = f), C) \equiv v}$$

$$\textsc{Alias}\ \frac{\mathbb{L}([x'] \mid\mid X, \textsc{Pred}(x), C) \equiv v}{\mathbb{L}([x] \mid\mid X, (x = x'), C) \equiv v}$$

$$\begin{array}{l}\textsc{Function}\\ \textsc{Enter}\\ \textsc{Parameter}\end{array}\ \frac{c = (x_r = x_f\ x_v) \qquad \mathbb{L}([x_v] \mid\mid X, \textsc{Pred}(c), C) \equiv v \qquad \mathbb{L}([x_f], \textsc{Pred}(c), C) \equiv [\texttt{fun}\ x\ \texttt{->}] \mid\mid e}{\mathbb{L}([x] \mid\mid X, (\texttt{fun}\ x\ \texttt{->}), [c] \mid\mid C) \equiv v}$$

$$\begin{array}{l}\textsc{Function}\\ \textsc{Enter}\\ \textsc{Non-Local}\end{array}\ \frac{\begin{array}{c}x'' \neq x \qquad c = (x_r = x_f\ x_v)\\ \mathbb{L}([x_f, x] \mid\mid X, \textsc{Pred}(c), C) \equiv v \qquad \mathbb{L}([x_f], \textsc{Pred}(c), C) \equiv [\texttt{fun}\ x''\ \texttt{->}] \mid\mid e\end{array}}{\mathbb{L}([x] \mid\mid X, (\texttt{fun}\ x''\ \texttt{->}), [c] \mid\mid C) \equiv v}$$

$$\textsc{Function Exit}\ \frac{\begin{array}{c}\mathbb{L}([x'] \mid\mid X, (x' = b), [\textsc{Cl}(x)] \mid\mid C) \equiv v\\ \textsc{RetCl}(e) = (x' = b) \qquad \mathbb{L}([x_f], \textsc{Pred}(c), C) \equiv [\texttt{fun}\ x''\ \texttt{->}] \mid\mid e\end{array}}{\mathbb{L}([x] \mid\mid X, (x = x_f\ x_v), C) \equiv v}$$

$$\textsc{Skip}\ \frac{x'' \neq x \qquad \mathbb{L}([x] \mid\mid X, \textsc{Pred}(x''), C) \equiv v \qquad \exists v_0.\ \mathbb{L}([x''], \textsc{Cl}(x''), C) \equiv v_0}{\mathbb{L}([x] \mid\mid X, (x'' = b), C) \equiv v}$$

From concrete to symbolic

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

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

From concrete to symbolic

- $\mathbb{L}([x_{f1}, x_{f2}, \dots, x], @x_{pp}, [\dots]) \equiv \mathbb{L}(\dots) \equiv \mathbb{L}(\dots) \equiv v$
  - 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
- $\mathbb{L}^{s}([x_{f1}, x_{f2}, \dots, x], @x_{pp}, [\dots]), \mathbb{L}(\dots), \mathbb{L}(\dots) \equiv {}^{s}v$ over Φ
  - Nondeterministic
    - Not know the call stack
    - Not know the input
  - Φ 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
  ret
```

- $\mathbb{L}$([fy], @fy, [])
  ≡ $\mathbb{L}$([fret], @fret, [fy])
  ≡ $\mathbb{L}$([x], @fret, [fy]) + 1
- $\mathbb{L}$([x], @fret, [fy])
  ≡ $\mathbb{L}$([x], @fun x->, [fy])
  ≡ $\mathbb{L}$([y], @fy, [])

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]), Φ^1
  $\equiv \mathbb{L}^s$([x], @fret, [fy]) + 1, Φ^2
- $\mathbb{L}^s$([x], @fret, [fy])
  $\equiv \mathbb{L}^s$([x], @fun x->, [fy]), Φ^3
  $\equiv \mathbb{L}^s$([y], @fy, []), Φ^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, [])

  ≡ $\mathbb{L}^s$([fret], @fret, [fy]), $\Phi^1 = \{\, {}^{[]}fy = {}^{[fy]}fret \,\}$

  ≡ $\mathbb{L}^s$([x], @fret, [fy]) + 1, $\Phi^2 = \{\, \ldots ,\, {}^{[fy]}ret = {}^{[fy]}x + 1 \,\}$

- $\mathbb{L}^s$([x], @fret, [fy])

  ≡ $\mathbb{L}^s$([x], @fun x->, [fy]), $\Phi^3 = \{\, \ldots \,\}$

  ≡ $\mathbb{L}^s$([y], @fy, []), $\Phi^4 = \{\, \ldots ,\, {}^{[fy]}x = {}^{[]}y \,\}$

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
```

- $\mathbb{L}^s([fy], @fy, [])$
  $\equiv \mathbb{L}^s([fret], @fret, [fy]),\ \Phi^1 = \{\ {}^{[]}fy = {}^{[fy]}fret\ \}$
  $\equiv \mathbb{L}^s([x], @fret, [fy]) + 1,\ \Phi^2 = \{\ \dots,\ {}^{[fy]}ret = {}^{[fy]}x + 1\ \}$
- $\mathbb{L}^s([x], @fret, [fy])$
  $\equiv \mathbb{L}^s([x], @\text{fun } x\text{->}, [fy]),\ \Phi^3 = \{\ \dots\ \}$
  $\equiv \mathbb{L}^s([y], @fy, []),\ \Phi^4 = \{\ \dots,\ {}^{[fy]}x = {}^{[]}y\ \}$ satisfiable, not interesting

Relative stack

```
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
```

- $\cancel{\mathbb{L}^s([fy], @fy, [])}$

  $\mathbb{L}^s([fret], @fret, [fy])$, $\Phi^1 = \{ {}^{[]}fy = {}^{[fy]}fret \}$

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

- $\mathbb{L}^s([x], @fret, [fy])$

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

  $\equiv \mathbb{L}^s([y], @fy, [])$, $\Phi^4 = \{ \ldots , {}^{[fy]}x = {}^{[]}y \}$ satisfiable, not interesting

Relative stack

```
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, $\Phi^2 = \{ \ldots , {}^{[]}ret = {}^{[]}x + 1 \}$
- $\mathbb{L}^s$([x], @fret, [])
  $\equiv \mathbb{L}^s$([x], @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) in
let fy = f y in
let f1 = f 1 in
let ret = fy + f1 in
  ret
```

- $\cancel{\mathbb{L}^s([fy], @fy, [])}$
  $\mathbb{L}^s([fret], @fret, [])$
  $\equiv \mathbb{L}^s([x], @fret, []) + 1$, $\Phi^2 = \{ \dots , {}^{[]}ret = {}^{[]}x + 1 \}$
- $\mathbb{L}^s([x], @fret, [])$
  $\equiv \mathbb{L}^s([x], @fun\ x\text{->}, [])$, $\Phi^3 = \{ \dots \}$
  $\equiv \mathbb{L}^s([y], @fy, [-fy])$, $\Phi^4 = \{ \dots , {}^{[]}x = {}^{[-fy]}y \}$

Relative stack

```
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
```

- $\mathbb{L}^s([fret], @fret, [])$

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

  $\equiv \mathbb{L}^s([x], @\text{fun } x\text{->}, [])$, $\Phi^3 = \{ \dots \}$

  $\cancel{\equiv \mathbb{L}^s([y], @fy, [-fy]), \Phi^4 = \{ \dots , {}^{[]}x = {}^{[-fy]}y \}}$ $\mathbb{L}^s([1], @fy, [-f1])$, $\Phi^4 = \{ \dots , {}^{[]}x = {}^{[-f1]}y \}$

Comparison of stacks

Stack type	**Concrete stack**	**Concrete stack**	**Relative stack**
Interpreter	Concrete	Concrete	Symbolic
Direction	Forward	Backward	Backward
Arrow	main -> target	main <- target	main <- target
Value at main entry	[ ]	[ ]	
Value at target	Can be non-empty	Can be non-empty	

Comparison of stacks

Stack type	**Concrete stack**	**Concrete stack**	**Relative stack**
Interpreter	Concrete	Concrete	Symbolic
Direction	Forward	Backward	Backward
Arrow	main -> target	main <- target	main <- target
Value at main entry	[ ]	[ ]	[-f1]
Value at target	Can be non-empty	Can be non-empty	[ ]

Comparison of stacks

Stack type	**Concrete stack**	**Concrete stack**	**Relative stack**
Interpreter	Concrete	Concrete	Symbolic
Direction	Forward	Backward	Backward
Arrow	main -> target	main <- target	main <- target
Value at main entry	[ ] as empty	[ ] as empty	[-f1]?[ ]
Value at target	Can be non-empty	Can be non-empty	[ ] as unknown

Comparison of stacks

Stack type	**Concrete stack**	**Concrete stack**	**Relative stack**
Interpreter	Concrete	Concrete	Symbolic
Direction	Forward	Backward	Backward
Arrow	main -> target	main <- target	main <- target
Value at main entry	[ ] as empty	[ ] as empty	[-f1]?[ ]
Value at target	Can be non-empty	Can be non-empty	[ ]?[ ]

Relative stack

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

(1) $\textsc{Push}([c_1, \ldots c_n]?[c'_1, \ldots c'_{n'}], c) = [c_1, \ldots c_n]?[c, c'_1, \ldots c'_{n'}]$,

(2) $\textsc{Pop}([c_1, \ldots c_n]?[], c) = [c, c_1, \ldots c_n]?[]$,

(3) $\textsc{Pop}([c_1, \ldots c_n]?[c'_1, \ldots c'_{n'}], c) = [c_1, \ldots c_n]?[c'_2, \ldots, c'_{n'}]$ *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) $\textsc{Concretize}(C?[]) = \textsc{Reverse}(C)$

- 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

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

From concrete to symbolic, formally

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

$^{\dot{C}}x$			*annotated vars*
$\mathcal{X}$			*annotated var sets*
$\dot{C}$	::=	$C?C$	*relative stacks*
ς	::=	$^{\dot{C}}x \mid \varsigma_{\mathtt{true}}$	*formulae symbols*
ϕ	::=	$\varsigma = \varsigma \odot \varsigma \mid \varsigma = \varsigma$	*formulae atoms*
		$\mid \varsigma = v \mid \mathtt{stack} = C$	
Φ	::=	$\phi \land \ldots \land \phi$	*formulae*
Π	::=	$\{\dot{C} \mapsto c, \ldots\}$	*search paths*

Fig. 7. New Constructs for Symbolic Lookup

From concrete to symbolic, formally

- $\mathbb{L}^s(X, \Phi, \Pi, c, \dot{C}) \equiv {}^s v$
  - X := lookup stack
  - Φ := constraint formulae
  - Π := search path
  - c ::= program point
  - $\dot{C}$ ::= relative stack
- Checking along the lookup
  - Φ is satisfiable
  - Π matches, a variable always points the same function in a nondeterministic trace
  - $\dot{C}$ has a empty normal stack part

$$
\begin{array}{lcll}
{}^{\dot{C}}x & & & \textit{annotated vars} \\
\mathcal{X} & & & \textit{annotated var sets} \\
\dot{C} & ::= & C?C & \textit{relative stacks} \\
\varsigma & ::= & {}^{\dot{C}}x \mid \varsigma_{\mathtt{true}} & \textit{formulae symbols} \\
\phi & ::= & \varsigma = \varsigma \odot \varsigma \mid \varsigma = \varsigma & \textit{formulae atoms} \\
 & & \mid \varsigma = v \mid \mathtt{stack} = C & \\
\Phi & ::= & \phi \wedge \ldots \wedge \phi & \textit{formulae} \\
\Pi & ::= & \{\dot{C} \mapsto c, \ldots\} & \textit{search paths}
\end{array}
$$

Fig. 7. New Constructs for Symbolic Lookup

Lookup rules, symbolically

$$\textsc{Function Enter Parameter}\quad \frac{\begin{array}{c}\dot{C}' = \textsc{Pop}(\dot{C}, c) \qquad \overset{S}{\mathbb{L}}([x_v] \,||\, X, \textsc{Pred}(c), \dot{C}') \equiv {}^{\dot{C}_0}x_0 \\ c = (x_r = x_f\ x_v) \qquad \Pi(\dot{C}) = c \qquad \overset{S}{\mathbb{L}}([x_f], \textsc{Pred}(c), \dot{C}') \equiv (\texttt{fun}\ x \texttt{->} (\ e\ ))\end{array}}{\overset{S}{\mathbb{L}}([x] \,||\, X, (\texttt{fun}\ x \texttt{->}), \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

$$\textsc{Function Enter Non-Local}\quad \frac{\begin{array}{c}\dot{C}' = \textsc{Pop}(\dot{C}, c) \qquad x'' \neq x \qquad c = (x_r = x_f\ x_v) \qquad \Pi(\dot{C}) = c \\ \overset{S}{\mathbb{L}}([x_f, x] \,||\, X, \textsc{Pred}(c), \dot{C}') \equiv {}^{\dot{C}_0}x_0 \qquad \overset{S}{\mathbb{L}}([x_f], \textsc{Pred}(c), \dot{C}') \equiv (\texttt{fun}\ x'' \texttt{->} (\ e\ ))\end{array}}{\overset{S}{\mathbb{L}}([x] \,||\, X, (\texttt{fun}\ x'' \texttt{->}), \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

$$\textsc{Function Exit}\quad \frac{\begin{array}{c}\overset{S}{\mathbb{L}}([x'] \,||\, X, (x' = b), \textsc{Push}(\dot{C}, \textsc{Cl}(x))) \equiv {}^{\dot{C}_0}x_0 \\ \textsc{RetCl}(e) = (x' = b) \qquad \overset{S}{\mathbb{L}}([x_f], \textsc{Pred}(x), \dot{C}) \equiv (\texttt{fun}\ x'' \texttt{->} (\ e\ ))\end{array}}{\overset{S}{\mathbb{L}}([x] \,||\, X, (x = x_f\ x_v), \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

$$\textsc{Skip}\quad \frac{x'' \neq x \qquad \overset{S}{\mathbb{L}}([x] \,||\, X, \textsc{Pred}(x''), \dot{C}) \equiv {}^{\dot{C}_0}x_0 \qquad \overset{S}{\mathbb{L}}([x''], \textsc{Cl}(x''), \dot{C}) \equiv \_}{\overset{S}{\mathbb{L}}([x] \,||\, X, (x'' = b), \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

Lookup rules, symbolically

$$\textsc{Value Discovery}\ \frac{\begin{array}{c}({}^{\dot{C}}x = v) \in \Phi \\ x \neq \textsc{FirstV}(e_{\mathrm{glob}}) \lor (\mathtt{stack} = \textsc{Concretize}(\dot{C})) \in \Phi \qquad \textsc{First}^{S}(x, c, \dot{C})\end{array}}{\overset{S}{\mathbb{L}}([x], (x = v), \dot{C}) \equiv {}^{\dot{C}}x}$$

$$\textsc{Input}\ \frac{\varsigma_{\mathtt{true}} = ({}^{\dot{C}}x = {}^{\dot{C}}x) \in \Phi \qquad x \neq \textsc{FirstV}(e_{\mathrm{glob}}) \lor (\mathtt{stack} = \textsc{Concretize}(\dot{C})) \in \Phi \qquad \textsc{First}^{S}(x, c, \dot{C})}{\overset{S}{\mathbb{L}}([x], (x = \mathtt{input}), \dot{C}) \equiv {}^{\dot{C}}x}$$

$$\textsc{Conditional Top}\ \frac{\begin{array}{c}\textsc{Cl}(x_1) = (x_1 = x_2\ ?\ e_{\mathtt{true}} : e_{\mathtt{false}}) \\ \overset{S}{\mathbb{L}}([x_2], \textsc{Pred}(x_1), \dot{C}) \equiv \beta \qquad \overset{S}{\mathbb{L}}(X, \textsc{Pred}(x_1), \dot{C}) \equiv {}^{\dot{C}_0}x_0\end{array}}{\overset{S}{\mathbb{L}}(X, (x_1\ !\ \beta), \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

$$\textsc{Conditional Bottom}\ \frac{\begin{array}{c}\overset{S}{\mathbb{L}}([x_2], \textsc{Pred}(x_1), \dot{C}) \equiv \beta \\ \overset{S}{\mathbb{L}}([x'] \,||\, X, (x' = b), \dot{C}) \equiv {}^{\dot{C}_0}x_0 \qquad \textsc{RetCl}(e_\beta) = (x' = b)\end{array}}{\overset{S}{\mathbb{L}}([x_1] \,||\, X, (x_1 = x_2\ ?\ e_{\mathtt{true}} : e_{\mathtt{false}}, \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

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