Higher-Order Demand-Driven Symbolic Evaluation

Zachary Palmer  Theodore Park  Scott F. Smith  Shiwei Weng
ICFP 2020, August 24-26, 2020

The Johns Hopkins University and Swarthmore College
## Forward vs Demand in Varying Domains

<table>
<thead>
<tr>
<th>System</th>
<th>Forward</th>
<th>Demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Programming</td>
<td>Forward-chain (uncommon)</td>
<td>Backward-chain</td>
</tr>
<tr>
<td>Tactic-based provers</td>
<td>Forward tactics (uncommon)</td>
<td>Goal-directed tactics</td>
</tr>
<tr>
<td>Program Analysis</td>
<td>(most are: (k)CFA etc)</td>
<td>Reps et al (imperative) DDPA (functional)</td>
</tr>
<tr>
<td>Symbolic Execution</td>
<td>(most are)</td>
<td>Snugglbug (imperative)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Here: <strong>DDSE</strong> (functional)</td>
</tr>
</tbody>
</table>
| Interpreter                   | (nearly all are) | Here: **DDI** (functional) \
|                               |               | *no substitution, environment or closures* |
1. The language syntax under study here
2. **DDI**, The novel demand-driven functional interpreter
3. **DDSE**, a demand-driven symbolic evaluator built on DDI
4. Implementation and evaluation of DDSE
Language Features in this Work

**In formal theory** functions, integers, booleans, conditionals, *input* (for test generation)

**Recursion** encoded via self-passing

**Also in implementation** recursive data structures

**ANF** Used to expose order of operations

  *e.g.* `let x = input in let y = x - 1 in let ret = x * y in ret`

**Unique variable names** a program point is named by its (unique) defining variable.
The DDI Lookup Function

- Basic idea follows programmer intuition: search upwards in code for variable definitions
- Lookup, \( \mathbb{L}([x], @x_{pp}, \_\_) \equiv v \), means \( x \) has value \( v \)
- \( @x_{pp} \) is the program point to begin (reverse) search from
- \( \_\_ \) is a call stack, of program call points.

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

- \( \mathbb{L}([y], @y, \_\_) \equiv 1 \)
The DDI Lookup Function

- Basic idea follows programmer intuition: search upwards in code for variable definitions
- Lookup, \( \mathbb{L}([x], @x_{pp}, \ldots) \equiv v \), means \( x \) has value \( v \)
- \( @x_{pp} \) is the program point to begin (reverse) search from
- \( \ldots \) is a call stack, of program call points.

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

- \( \mathbb{L}([y], @y, \ldots) \equiv 1 \)
- \( \mathbb{L}([y], @f1, \ldots) \equiv 1 \)
The DDI Lookup Function

- Basic idea follows programmer intuition: search upwards in code for variable definitions
- Lookup, \( L([x], \@x_{pp}, \_\_\_) \equiv v \), means \( x \) has value \( v \)
- \( \@x_{pp} \) is the program point to begin (reverse) search from
- \( \_\_\_ \) is a call stack, of program call points.

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

- \( L([y], \@y, \_\_\_) \equiv 1 \)
- \( L([y], \@f1, \_\_\_) \equiv 1 \)
- \( L([x], \@\text{fret}, f1) \equiv 1 \)
Tracing Function Application

Function application requires call-return alignment

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

1. \( L([f1], @f1, \_\_\_\_\_\_) \)
Tracing Function Application

Function application requires call-return alignment

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

1. \( \text{L}([f1], @f1, \_\_\_\_ \))
2. \( \equiv \text{L}([\text{fret}], @\text{fret}, f1) \)
Function application requires call-return alignment

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

1. $L([f1], @f1, \_)$
2. $\equiv L([fret], @fret, f1)$
3. $\equiv L([x], @fun\ x, f1) + 1$
Function application requires call-return alignment

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

1. \( L([f1], @f1, \_\_\_) \)
2. \( \equiv L([fret], @fret, f1) \)
3. \( \equiv L([x], @fun x, f1) + 1 \)
4. \( L([x], @fun x, f1) \equiv L([y], @f1, \_\_\_) \)
Function application requires call-return alignment

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

1. \( L([f1], @f1, \_\_\_\_) \)
2. \( \equiv L([fret], @fret, f1) \)
3. \( \equiv L([x], @fun \ x, f1) + 1 \)
4. \( L([x], @fun \ x, f1) \equiv L([y], @f1, \_\_\_) \)
5. \( L([y], @f1, \_\_\_) \equiv 1 \) so final result is \( L([f1], @f1, \_\_\_) \equiv 2. \)
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
let g =
    (fun x ->
        let gret = (fun y ->
            let gyret = x + y in gyret) in gret)
    in
define g5 = g 5
let ret = g5 1 in ret

1. ... \( L(\{x, \gamma_{gyret}, ret\}) \equiv L(\{g5, x\}, \gamma_{g5}, ...) \):
   1.1 find definition site for \( g5 \);
   1.2 then, resume search for \( x \) since that is lexical scope of its def’n.
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

1. \( \mathbb{L}([x], \circ gyret, \underline{\text{ret}}) \equiv \mathbb{L}([g5, x], \circ g5, \underline{\text{g5}}) \):
   1.1 find definition site for g5;
   1.2 then, resume search for x since that is lexical scope of its def'n.

2. \( \mathbb{L}([g5, x], \circ g5, \underline{\text{g5}}) \equiv \mathbb{L}([gret, x], \circ gret, \underline{\text{g5}}) \)
let g =
    (fun x ->
        let gret = (fun y ->
            let gyret = x + y in gyret) in gret) in gret

let g5 = g 5 in
let ret = g5 1 in ret

1. ... \( L([x, \@gyret, ret]) \equiv L([g5, x, \@g5, \_]) \):
   1.1 find definition site for \( g5 \);
   1.2 then, resume search for \( x \) since that is lexical scope of its def'n.
2. \( L([g5, x, \@g5, \_]) \equiv L([gret, x, \@gret, g5]) \)
3. \( L([gret, x, \@gret, g5]) \equiv L([x, \@fun x, g5]) \)
Non-Local Variables

```ocaml
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
```

1. \[ \mathcal{L}([x], @gyret, ret) \equiv \mathcal{L}([g5, x], @g5, \_): \]
   1.1 find definition site for \( g5 \);
   1.2 then, resume search for \( x \) since that is lexical scope of its def'n.
2. \[ \mathcal{L}([g5, x], @g5, \_) \equiv \mathcal{L}([gret, x], @gret, g5) \]
3. \[ \mathcal{L}([gret, x], @gret, g5) \equiv \mathcal{L}([x], @fun x, g5) \]
4. \[ \mathcal{L}([x], @fun x, g5) \equiv 5 \]
Non-Local Variables

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

1. ...LL([x, @gyret, ret]) ≡ LL([g5, x], @g5, ___):
    1.1 find definition site for g5;
    1.2 then, resume search for x since that is lexical scope of its def'n.
2. LL([g5, x], @g5, ___) ≡ LL([gret, x], @gret, g5)
3. LL([gret, x], @gret, g5) ≡ LL([x], @fun x, g5)
4. LL([x], @fun x, g5) ≡ 5

General Lookup signature: LL([x_{f_1}, \ldots, x_{f_n}, x], @x_{pp}, ___) ≡ v.
**Peek at Full Rules for Functional Core**

<table>
<thead>
<tr>
<th>Value Discovery</th>
<th>Value Discard</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{First}(x, \ Cl(x), C) )</td>
<td>( \mathbb{L}([x], (x = v), C) \equiv v )</td>
</tr>
<tr>
<td>( \mathbb{L}([x]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alias</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{L}([x']</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function Enter (Parameter)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( c = (x_r = xf \ x_v) )</td>
<td>( \mathbb{L}([x_v]</td>
</tr>
<tr>
<td>( \mathbb{L}([x]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function Enter (Non-Local)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x'' \neq x ) ( c = (x_r = xf \ x_v) )</td>
<td>( \mathbb{L}([x_f, x]</td>
</tr>
<tr>
<td>( \mathbb{L}([x]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function Exit</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{RetCl}(e) = (x' = b) )</td>
<td>( \mathbb{L}([x_f], \ Pred(c), C) \equiv [\text{fun } x'' \rightarrow]</td>
</tr>
<tr>
<td>( \mathbb{L}([x]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Skip</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x'' \neq x )</td>
<td>( \mathbb{L}([x]</td>
</tr>
<tr>
<td>( \exists v_0. \mathbb{L}([x''], \ Cl(x''), C) \equiv v_0 )</td>
<td>( \mathbb{L}([x]</td>
</tr>
</tbody>
</table>
Symbolic lookup: $L^S([x_{f_1}, \ldots, x_{f_n}, x], \@x_{pp}, \ldots) \equiv \exists \bar{x}$ over $\Phi$

- Lookup returns a variable activation now: a pair $\bar{x}$
- $\Phi$ equationally constrains variables, must be satisfiable

```ocaml
let y = input in
let f =
    (fun x ->
      let fret = x + 1 in fret)
let f1 = f y in ret
let ret = f f1 in ret
```
Symbolic lookup: $\llbracket S \rrbracket ([x_{f_1}, \ldots, x_{f_n}, x], \llbracket @x_{pp}, x \rrbracket) \equiv \llbracket x \rrbracket \text{ over } \Phi$

- Lookup returns a variable activation now: a pair $\llbracket x \rrbracket$
- $\Phi$ equationally constrains variables, must be satisfiable

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

1. $\llbracket S \rrbracket ([f1], @f1, \_)$
From Demand Interpreter to Demand Symbolic Evaluation

Symbolic lookup: \( L^S([x_{f_1}, \ldots, x_{f_n}, x], @xp_p, \_\_\_) \equiv L^S(x) \) over \( \Phi \)

- Lookup returns a variable activation now: a pair \( L^S(x) \)
- \( \Phi \) equationally constrains variables, must be satisfiable

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

1. \( L^S([f1], @f1, \_\_\_) \equiv L^S([fret], @fret, f1) \)
From Demand Interpreter to Demand Symbolic Evaluation

Symbolic lookup: $\mathbb{L}^S([x_{f_1}, \ldots, x_{f_n}, x], \mathbb{S}_{x_{pp}, \_}) \equiv \mathbb{L}^S_x$ over $\Phi$

- Lookup returns a variable activation now: a pair $\mathbb{L}^S_x$
- $\Phi$ equationally constrains variables, must be satisfiable

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

1. $\mathbb{L}^S([f_1], \_f_1, \_f_1) \equiv \mathbb{L}^S([fret], \_fret, \_f_1)$

2. $\equiv \var{f_1}fret; (\var{f_1}fret = \mathbb{L}([x], \mathbb{S}_{\text{fun } x, \_f_1}) + 1) \in \Phi$
Symbolic lookup: \( \mathbb{L}_S([x_{f_1}, \ldots, x_{f_n}, x], \otimes x_{pp}, \cdot) \equiv \cdot \cdot x \) over \( \Phi \)

- Lookup returns a variable activation now: a pair \( \cdot \cdot x \)
- \( \Phi \) equationally constrains variables, must be satisfiable

```ocaml
define y = input
let f =
(\( x \rightarrow 
    let fret = x + 1 \) in fret)
in
let f1 = f y in ret
let ret = f f1 in ret
```

1. \( \mathbb{L}_S([f1], \otimes f1, \cdot) \equiv \mathbb{L}_S([fret], \otimes fret, f1) \)
2. \( \equiv f1 \cdot fret; (f1 \cdot fret = \mathbb{L}([x], \otimes fun x, f1) + 1) \in \Phi \)
3. \( \mathbb{L}([x], \otimes fun x, f1) \equiv \mathbb{L}([y], f, \cdot) \)
Symbolic lookup: $\llbracket S([x_{f_1}, \ldots, x_{f_n}, x], @x_p, \_\_\_\_) \equiv \_\_\_\_x$ over $\Phi$

- Lookup returns a variable activation now: a pair $\_\_\_\_x$
- $\Phi$ equationally constrains variables, must be satisfiable

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

1. $\llbracket S([f1], @f1, \_\_\_\_) \equiv \llbracket S([fret], @fret, f1)\r
2. $\equiv f1\_fret; (f1\_fret = \llbracket S([x], @fun x, f1) + 1) \in \Phi$
3. $\llbracket S([x], @fun x, f1) \equiv \llbracket S([y], f, \_\_\_\_)$
4. $\equiv \llbracket S([y], @f, \_\_\_\_)$

5. Final $\Phi = \{f1\_fret = y + 1\}$; satisfiable.
Symbolic lookup: $L^S([x_{f_1}, \ldots, x_{f_n}, x], \mathcal{O}x_{pp}, \mathcal{X}) \equiv \mathcal{X}$ over $\Phi$

- Lookup returns a variable activation now: a pair $\mathcal{X}$
- $\Phi$ equationally constrains variables, must be satisfiable

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

1. $L^S([f1], \mathcal{O}f1, \mathcal{X}) \equiv L^S([fret], \mathcal{O}fret, f1)$
2. $\equiv f1 \cdot fret; \ (f1 \cdot fret = L([x], \mathcal{O}fun x, f1) + 1) \in \Phi$
3. $L([x], \mathcal{O}fun x, f1) \equiv L([y], f, \mathcal{X})$
4. $\equiv L([y], \mathcal{O}f, \mathcal{X}) \equiv L([y], \mathcal{O}y, \mathcal{X}) \equiv y$
5. Final $\Phi = \{ f1 \cdot fret = y + 1 \};$ satisfiable.
Formal Development

- **Theorem:** Demand operational semantics $\equiv$ Forward operational semantics

- **Theorem:** DDSE is sound and complete with respect to operational semantics

- Several subtle issues had to be skipped in talk:
  1. Call stack must be inferred when lookup initiated in middle of program
  2. Input order of demand-driven lookup is not forward order; sorting step needed
DDSE 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
- Need to dovetail on different search paths
  \[ \Rightarrow \] coroutine/nondeterminism monad used
- Successfully solves all benchmarks from Cruanes [CADE '17], a higher-order forward symbolic evaluator implementation; see paper for details
Comparison with Select Related Work

- Snugglebug, PLDI ’09: *Imperative* demand symbolic execution, no correctness
- Cruanes, 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 framework implementation; bounded datatypes only
- This work: *functional*, *demand*, *arbitrary* datatypes and recursion, *proven* sound and complete.