# Toward a Live Stepper for Typed Expressions with Holes


**Cyrus Omar** The University of Chicago

Ian Voysey Carnegie Mellon University

Matthew A. Hammer University of Colorado Boulder



LIVE 2017 – Oct. 24, 2017

Live programming environments
**"narrow the temporal and perceptive gap**
**between program development and code execution"**.

[Burckhardt et al. *It’s alive! continuous feedback in UI programming*. PLDI 2013]

Live programming environments
**"narrow the temporal and perceptive gap**
**between program development and code execution"**.

We know how to execute **complete** (i.e. well-formed + well-typed) **terms**...

[Burckhardt et al. *It's alive! continuous feedback in UI programming*. PLDI 2013]

Live programming environments
**"narrow the temporal and perceptive gap**
**between program development and code execution"**.

...but during program development, we often encounter **incomplete** terms.

[Burckhardt et al. *It's alive! continuous feedback in UI programming*. PLDI 2013]

# Example: Klipse

http://blog.klipse.tech/ocaml/2017/10/05/blog-ocaml.html

# Incomplete Programs → Programs with Holes

```
let rec fac n =

let test = List.map fac [1; 2; 3; 4; 5]
```

# Incomplete Programs → Programs with Holes

```
let rec fac n = ␣a

let test = List.map fac [1; 2; 3; 4; 5]
```

# Incomplete Programs → Programs with Holes

```
let rec fac n = ␣a

let test = List.map fac [1; 2; 3; 4; 5]
```

- Error recovery [de Jonge et al., SLE 2009; Kats et al., OOPSLA 2009]
- Programmer inserts explicitly [GHC, Agda, others]
- Editor often inserts explicitly [Amorim et al., SLE 2016]
- Editor always inserts (i.e. a structure editor) [Omar et al., POPL 2017]

# Incomplete Programs → Programs with Holes

```
let rec fac n = ␣ᵃ

let test = List.map 0ᵇ [1; 2; 3; 4; 5]
```

Type inconsistencies are non-empty holes!
[Omar et al., POPL 2017]

# Running programs with holes

```
let rec fac n = ␣ᵃ

let test = List.map fac [1; 2; 3; 4; 5]
```

# Running programs with holes

```
let rec fac n = ␣a

let test = List.map fac [1; 2; 3; 4; 5]
```

```
[␣a.1; ␣a.2; ␣a.3; ␣a.4; ␣a.5]
```

# Running programs with holes

```
let rec fac n = ␣a

let test = List.map fac [1; 2; 3; 4; 5]
```

```
[␣a.1; ␣a.2; ␣a.3; ␣a.4; ␣a.5]
```

# Running programs with holes

```
let rec fac n = ␣a

let test = List.map fac [1; 2; 3; 4; 5]
```

```
[␣a.1; ␣a.2; ␣a.3; ␣a.4; ␣a.5]
```

```
Hole a
n : int
  1   @a.1
  2   @a.2
  3   @a.3
  4   @a.4
  5   @a.5
fac : int → ␣
  [fun fac…] @a.*
```

# Running programs with holes

```
let rec fac n =
  match n with
  | 1 → 1
  | _ →
    let pred = n – 1 in ␣a

let test = List.map fac [1; 2; 3; 4; 5]
```

```
 [1; ␣a.1; ␣a.2; ␣a.3; ␣a.4]
```

```
Hole a
pred : int
  1  @a.1
  2  @a.2
  …
n : int
  2  @a.1
  3  @a.2
  …
fac : int → ␣
  [fun fac…] @a.*
```

# Running programs with holes

```
let rec fac n =
  match n with
  | 1 → 1
  | _ →
    let pred = n - 1 in ␣a

let test = List.map fac [␣b; 2; 3; 4; 5]
```

```
 [match ␣b.1 with
  | 1 → 1
  | _ →
    let pred = ␣b.2 - 1 in ␣a.1;
 ␣a.2; ␣a.3; ␣a.4; ␣a.5]
```

```
Hole a
pred : int
  1   @a.2
  2   @a.3
  …
n : int
  ␣b  @a.1
  2   @a.2
  …
fac : int → ␣
  [fun fac…] @a.*
```

# Semantics

$$
\begin{array}{rrcl}
\text{HTyp} & \tau & ::= & b \mid \tau \to \tau \mid (\!|\,|\!) \\
\text{HExp} & e & ::= & c \mid x \mid \lambda x{:}\tau.e \mid e(e) \mid (\!|\,|\!)^u \mid (\!|e|\!)^u \mid \lambda x.e \mid e : \tau \\
\text{DHExp} & d & ::= & c \mid x \mid \lambda x{:}\tau.d \mid d(d) \mid (\!|\,|\!)^u_\sigma \mid (\!|d|\!)^u_\sigma \mid \langle\tau\rangle\, d
\end{array}
$$

Hole environments (n-ary substitutions) – borrowed from **context modal type theory** [Nanevski, Pientka and Pfenning, TOCL 2007]

# Semantics

$$
\begin{array}{llll}
\text{HTyp} & \tau & ::= & b \mid \tau \to \tau \mid (\!|\,|\!) \\
\text{HExp} & e & ::= & c \mid x \mid \lambda x{:}\tau.e \mid e(e) \mid (\!|\,|\!)^u \mid (\!|e|\!)^u \mid \lambda x.e \mid e : \tau \\
\text{DHExp} & d & ::= & c \mid x \mid \lambda x{:}\tau.d \mid d(d) \mid (\!|\,|\!)^u_\sigma \mid (\!|d|\!)^u_\sigma \mid \langle \tau \rangle\, d
\end{array}
$$

Dynamic casts – borrowed from **gradual type theory**
[Siek and Taha, 2006]

# Semantics

| | Scenario 1: Initial Stepping | | Scenario 2: Edit and Resume |
|---|---|---|---|
| | fun $f(x, y) = 3 + x * y \div \bigcirc^{u}_{[x/x, y/y]} + 2 * x$ | $\xrightarrow{[\![(x+1)/u]\!]}$ | fun $f(x, y) = 3 + x * y \div (x + 1) + 2 * x$ |
| 1 | $f(2, 3) \longmapsto 3 + 2 * 3 \div \bigcirc^{u}_{[2/x, 3/y]} + 2 * 2$ | $\xrightarrow{[\![(x+1)/u]\!]}$ | $f(2, 3) \longmapsto 3 + 2 * 3 \div (2 + 1) + 2 * 2$ |
| 2 | $\longmapsto 3 + 6 \div \bigcirc^{u}_{[2/x, 3/y]} + 2 * 2$ | $\xrightarrow{[\![(x+1)/u]\!]}$ | $\longmapsto 3 + 6 \div (2 + 1) + 2 * 2$ |
| 3 | $\longmapsto 3 + 6 \div \bullet^{u}_{[2/x, 3/y]} + 2 * 2$ | $\xrightarrow{[\![(x+1)/u]\!]}$ | $\longmapsto 3 + 6 \div 3 + 2 * 2$ |
| 4 | | | $\longmapsto 3 + 2 + 2 * 2$ |
| 5 | | | $\longmapsto 5 + 2 * 2$ |
| 6 | $\longmapsto 3 + 6 \div \bullet^{u}_{[2/x, 3/y]} + 4$ | $\xrightarrow{[\![(x+1)/u]\!]} \longmapsto^{*}$ | $\longmapsto 5 + 4$ |
| 7 | | | $\longmapsto 9$ |

# Semantics

| | Scenario 1: Initial Stepping | | Scenario 2: Edit and Resume |
|---|---|---|---|
| | $\mathsf{fun}\ f(x,y) = 3 + x * y \div \bigcirc^{u}_{[x/x,y/y]} + 2 * x$ | $\xrightarrow{[\![(x+1)/u]\!]}$ | $\mathsf{fun}\ f(x,y) = 3 + x * y \div (x+1) + 2 * x$ |
| 1 | $f(2,3) \longmapsto 3 + 2 * 3 \div \bigcirc^{u}_{[2/x,3/y]} + 2 * 2$ | $\xrightarrow{[\![(x+1)/u]\!]}$ | $f(2,3) \longmapsto 3 + 2 * 3 \div (2+1) + 2 * 2$ |
| 2 | $\longmapsto 3 + 6 \div \bigcirc^{u}_{[2/x,3/y]} + 2 * 2$ | $\xrightarrow{[\![(x+1)/u]\!]}$ | $\longmapsto 3 + 6 \div (2+1) + 2 * 2$ |
| 3 | $\longmapsto 3 + 6 \div \bullet^{u}_{[2/x,3/y]} + 2 * 2$ | $\xrightarrow{[\![(x+1)/u]\!]}$ | $\longmapsto 3 + 6 \div 3 + 2 * 2$ |
| 4 | | | $\longmapsto 3 + 2 + 2 * 2$ |
| 5 | | | $\longmapsto 5 + 2 * 2$ |
| 6 | $\longmapsto 3 + 6 \div \bullet^{u}_{[2/x,3/y]} + 4$ | $\xrightarrow{[\![(x+1)/u]\!]} \longmapsto^{*}$ | $\longmapsto 5 + 4$ |
| 7 | | | $\longmapsto 9$ |

A notion of **type safety** that can handle evaluation states that are neither values nor steppable (i.e. indeterminate)

# Semantics

| | Scenario 1: Initial Stepping | | Scenario 2: Edit and Resume |
|---|---|---|---|
| | fun $f(x,y) = 3 + x * y \div \bigcirc^{u}_{[x/x,y/y]} + 2 * x$ | $\xrightarrow{[\![(x+1)/u]\!]}$ | fun $f(x,y) = 3 + x * y \div (x+1) + 2 * x$ |
| 1 | $f(2,3) \longmapsto 3 + 2 * 3 \div \bigcirc^{u}_{[2/x,3/y]} + 2 * 2$ | $\xrightarrow{[\![(x+1)/u]\!]}$ | $f(2,3) \longmapsto 3 + 2 * 3 \div (2+1) + 2 * 2$ |
| 2 | $\longmapsto 3 + 6 \div \bigcirc^{u}_{[2/x,3/y]} + 2 * 2$ | $\xrightarrow{[\![(x+1)/u]\!]}$ | $\longmapsto 3 + 6 \div (2+1) + 2 * 2$ |
| 3 | $\longmapsto 3 + 6 \div \bullet^{u}_{[2/x,3/y]} + 2 * 2$ | $\xrightarrow{[\![(x+1)/u]\!]}$ | $\longmapsto 3 + 6 \div 3 + 2 * 2$ |
| 4 | | | $\longmapsto 3 + 2 + 2 * 2$ |
| 5 | | | $\longmapsto 5 + 2 * 2$ |
| 6 | $\longmapsto 3 + 6 \div \bullet^{u}_{[2/x,3/y]} + 4$ | $\xrightarrow{[\![(x+1)/u]\!]}\longmapsto^{*}$ | $\longmapsto 5 + 4$ |
| 7 | | | $\longmapsto 9$ |

A notion of **hole instantiation** and a **commutativity theorem** that allows edits to holes to continue from the previous evaluation state, rather than restart on each edit.

# Demo: Hazel

http://hazel.org/hazel/hazel.html

# Conclusion



**Incomplete programs** arise frequently.

**Holes** make reasoning about incomplete programs possible.

We can **run programs with holes** until the hole ends up in elimination position.

We can **track the environment around each hole instance**.

The **semantics** are rooted in gradual typing (for type holes) and contextual modal logic (for expression holes), and there are some interesting and non-trivial theorems!

**Future work**: finish proofs, UI design (esp. for a single stepper), empirical evaluation, how to handle effects, blame