# Practical performance enhancements to the evaluation model of the Hazel programming environment

Jonathan Lam[1] Prof. Fred Fontaine, Advisor[1]
Prof. Robert Marano, Co-advisor[1] Prof. Cyrus Omar[2]

[1]Electrical Engineering
The Cooper Union for the Advancement of Science and Art

[2]Electrical Engineering and Computer Science
Future of Programming Lab (FPLab), University of Michigan

2022/04/29

# Overview I

Project context

Implementation-based Mostly practically-driven

Functional programming Context for PL theory

Hazel live programming environment An experimental editor with typed holes aimed at solving the "gap problem," developed at UM

# Overview II

Project scope

**Evaluation with environments** Lazy variable lookup for performance

**Hole instances to hole closures** Redefining hole instances for performance

**Implementing fill-and-resume (FAR)** Efficiently resume evaluation

Project evaluation

**Empirical evaluation** Measure performance gain of motivating cases

**Informal metatheory** State metatheorems and provide proof sketches

# Table of Contents


1. Primer on PL theory
2. The Hazel live programming environment
3. Evaluation using the environment model
4. Identifying hole instances by physical environment
5. The fill-and-resume (FAR) optimization
6. Empirical results
7. Theoretical results/innovations
8. Future work/conclusions

# A programming language is a specification

**Syntax** is the grammar of a valid program

**Semantics** describes the behavior of a syntactically valid program

$$\tau ::= \tau \to \tau \mid b \mid (\!|\,|\!)$$
$$e ::= c \mid x \mid \lambda x : \tau . e \mid e\ e \mid e : \tau \mid (\!|\,|\!) \mid (\!| e |\!)$$

Figure: Hazelnut grammar

# Static and dynamic semantics

**Statics** Edit actions, type-checking, elaboration ("compile-time")

**Dynamics** Evaluation ("run-time")

$$\frac{e_1 \Downarrow \lambda x.e_1' \qquad e_2 \Downarrow e_2' \qquad [e_2'/x]e_1' = e}{e_1\ e_2 \Downarrow e}\ \text{EAp}$$

Figure: Evaluation rule for function application using a big-step semantics

# A brief primer on the $\lambda$-calculus

Untyped $\lambda$-calculus Simple universal model of computation by Church

Simply-typed $\lambda$-calculus Extension of the ULC with static type-checking

Gradually-typed $\lambda$-calculus Optionally-typed, with "pay-as-you-go" benefits of static typing

$$
\begin{aligned}
e ::= {} & x \\
\mid {} & \lambda x.e \\
\mid {} & e\ e
\end{aligned}
$$

(a) Grammar

$$
\frac{}{\lambda x.e \Downarrow \lambda x.e}\ \Lambda\text{-ELam}
$$

$$
\frac{e_1 \Downarrow \lambda x.e_1' \qquad [e_2/x]e_1' \Downarrow e}{e_1\ e_2 \Downarrow e}\ \Lambda\text{-EAp}
$$

(b) Dynamic semantics

Figure: The untyped $\lambda$-calculus

# Table of Contents


1. Primer on PL theory
2. The Hazel live programming environment
3. Evaluation using the environment model
4. Identifying hole instances by physical environment
5. The fill-and-resume (FAR) optimization
6. Empirical results
7. Theoretical results/innovations
8. Future work/conclusions

# The Hazel programming language and environment

- Live programming: Rapid static and dynamic feedback (“gap problem”)
- Structured editor: Elimination of syntax errors
- Bidirectionally typed: Simple type inference
- Gradually typed: Hole type and cast-calculus based on Siek et al. [1, 2]
- Purely functional: Avoids side-effects and promotes commutativity


(a) The Hazelgrove organization

(b) Implemented in ReasonML and JSOO

Figure: Hazel implementation

# The Hazel programming interface

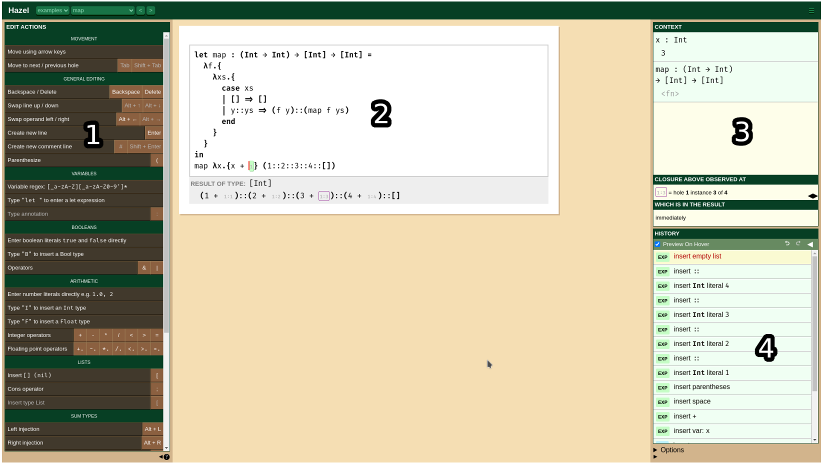



Figure: The Hazel interface

# Hazelnut: A bidirectionally-typed static semantics

**(Typed) expression holes** Internalize “red squiggly underlines”

**Action semantics** Structural editing behavior, ensures always well-typed

| # | Z-Expression | Next Action | Rule |
|---|---|---|---|
| 1 | $\triangleright\llparenthesis\rrparenthesis\triangleleft$ | `construct lam` $x$ | (13e) |
| 2 | $(\lambda x.\llparenthesis\rrparenthesis) : (\triangleright\llparenthesis\rrparenthesis\triangleleft \to \llparenthesis\rrparenthesis)$ | `construct num` | (12b) |
| 3 | $(\lambda x.\llparenthesis\rrparenthesis) : (\triangleright\mathtt{num}\triangleleft \to \llparenthesis\rrparenthesis)$ | `move parent` | (6c) |
| 4 | $(\lambda x.\llparenthesis\rrparenthesis) : \triangleright(\mathtt{num} \to \llparenthesis\rrparenthesis)\triangleleft$ | `move child 2` | (6b) |
| 5 | $(\lambda x.\llparenthesis\rrparenthesis) : (\mathtt{num} \to \triangleright\llparenthesis\rrparenthesis\triangleleft)$ | `construct num` | (12b) |
| 6 | $(\lambda x.\llparenthesis\rrparenthesis) : (\mathtt{num} \to \triangleright\mathtt{num}\triangleleft)$ | `move parent` | (6d) |
| 7 | $(\lambda x.\llparenthesis\rrparenthesis) : \triangleright(\mathtt{num} \to \mathtt{num})\triangleleft$ | `move parent` | (8d) |
| 8 | $\triangleright(\lambda x.\llparenthesis\rrparenthesis) : (\mathtt{num} \to \mathtt{num})\triangleleft$ | `move child 1` | (8a) |
| 9 | $\triangleright(\lambda x.\llparenthesis\rrparenthesis)\triangleleft : (\mathtt{num} \to \mathtt{num})$ | `move child 1` | (8e) |
| 10 | $(\lambda x.\triangleright\llparenthesis\rrparenthesis\triangleleft) : (\mathtt{num} \to \mathtt{num})$ | `construct var` $x$ | (13c) |
| 11 | $(\lambda x.\triangleright x\triangleleft) : (\mathtt{num} \to \mathtt{num})$ | `construct plus` | (13l) |
| 12 | $(\lambda x.(x + \triangleright\llparenthesis\rrparenthesis\triangleleft)) : (\mathtt{num} \to \mathtt{num})$ | `construct lit 1` | (13j) |
| 13 | $(\lambda x.(x + \triangleright\underline{1}\triangleleft)) : (\mathtt{num} \to \mathtt{num})$ | — | — |

**Figure 1.** Constructing the increment function in Hazelnut.

now assume $incr : \mathtt{num} \to \mathtt{num}$

| # | Z-Expression | Next Action | Rule |
|---|---|---|---|
| 14 | $\triangleright\llparenthesis\rrparenthesis\triangleleft$ | `construct var` *incr* | (13c) |
| 15 | $\triangleright incr\triangleleft$ | `construct ap` | (13h) |
| 16 | $incr(\triangleright\llparenthesis\rrparenthesis\triangleleft)$ | `construct var` *incr* | (13d) |
| 17 | $incr(\llparenthesis\triangleright incr\triangleleft\rrparenthesis)$ | `construct ap` | (13h) |
| 18 | $incr(\llparenthesis incr(\triangleright\llparenthesis\rrparenthesis\triangleleft)\rrparenthesis)$ | `construct lit 3` | (13j) |
| 19 | $incr(\llparenthesis incr(\triangleright\underline{3}\triangleleft)\rrparenthesis)$ | `move parent` | (8j) |
| 20 | $incr(\llparenthesis\triangleright incr(\underline{3})\triangleleft\rrparenthesis)$ | `move parent` | (8p) |
| 21 | $incr(\triangleright\llparenthesis incr(\underline{3})\rrparenthesis\triangleleft)$ | `finish` | (16b) |
| 22 | $incr(\triangleright incr(\underline{3})\triangleleft)$ | — | — |

**Figure 2.** Applying the increment function.

Figure: Sample Hazelnut action sequence [3]

# Hazelnut Live: A bidirectionally-typed dynamic semantics

**Internal language** Cast calculus from Siek et al. [1, 2] for dynamic typing

**Hole evaluation** Evaluation continues *around* holes, captures environment

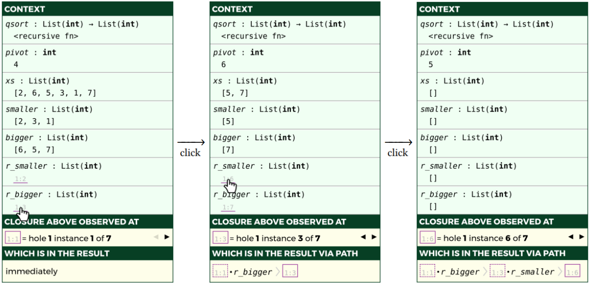



Figure: Illustration of Hazelnut Live context inspector [4]

# Table of Contents


1. Primer on PL theory
2. The Hazel live programming environment
3. Evaluation using the environment model
4. Identifying hole instances by physical environment
5. The fill-and-resume (FAR) optimization
6. Empirical results
7. Theoretical results/innovations
8. Future work/conclusions

# Evaluation using environments vs. substitution

**[TODO: comparison table, illustration of how each one works]**

# Updated evaluation rules

$\boxed{\sigma \vdash d \Downarrow d'}$ $d$ evaluates to $d'$ given environment $\sigma$

$$\frac{}{\sigma \vdash (\lambda x : \tau . d) \Downarrow [\sigma](\lambda x : \tau . d')}\ \text{ELam}$$

$$\frac{}{\sigma, x \leftarrow d \vdash x \Downarrow d}\ \text{EVar}$$

$$\frac{\sigma \vdash d_1 \Downarrow [\sigma'] \lambda x : \tau . d_1' \qquad \sigma \vdash d_2 \Downarrow d_2' \qquad \sigma', x \leftarrow d_2' \vdash d_1' \Downarrow d}{\sigma \vdash d_1\ d_2 \Downarrow d}\ \text{EAp}$$

$$\frac{}{\sigma \vdash (\!|\!)^u \Downarrow [\sigma](\!|\!)^u}\ \text{EvalB-EHole} \qquad \frac{\sigma \vdash d \Downarrow d'}{\sigma \vdash (\!| d |\!)^u \Downarrow [\sigma](\!| d' |\!)^u}\ \text{EvalB-NEHole}$$

Figure: Big-step semantics for evaluation with environments

# Handling recursion

**Fixpoint form** Useful for a pure implementation of recursive functions, from Plotkin's System PCF

$$\frac{\sigma \vdash d \Downarrow [\sigma']d'}{\sigma \vdash \mathrm{fix}\, f : \tau.d \Downarrow [\sigma, f \leftarrow \mathrm{fix}\, f : \tau.[\sigma']d']d'}\ \text{EFix}$$

$$\frac{d \neq \mathrm{fix}\, f : \tau.d'}{\sigma, x \leftarrow d \vdash x \Downarrow d}\ \text{EVar} \qquad \frac{\sigma \vdash \mathrm{fix}\, f : \tau.d \Downarrow d'}{\sigma, x \leftarrow \mathrm{fix}\, f : \tau.d \vdash x \Downarrow d'}\ \text{EUnwind}$$

Figure: Big-step semantics for evaluation of fixpoints

# Matching the result from evaluation using substitution

$\boxed{d \Uparrow_{[]} d'}$ $d$ is substitutes to $d'$ inside the evaluation boundary

$$\frac{}{c \Uparrow_{[]} c}\ \text{PPI}_{[]}\text{Const} \qquad \frac{d_1 \Uparrow_{[]} d_1' \qquad d_2 \Uparrow_{[]} d_2'}{d_1\ d_2 \Uparrow_{[]} d_1'\ d_2'}\ \text{PPI}_{[]}\text{Ap} \qquad \frac{\sigma \Uparrow_{[]} \sigma' \qquad \sigma' \vdash d \Uparrow_{[]} d'}{[\sigma]d \Uparrow_{[]} d'}\ \text{PPI}_{[]}\text{Closure}$$

$\boxed{\sigma \vdash d \Uparrow_{[]} d'}$ $d$ substitutes to $d'$ outside the evaluation boundary

$$\frac{}{c \Uparrow_{[]} c}\ \text{PPO}_{[]}\text{Const} \qquad \frac{}{\sigma, x \leftarrow d \vdash x \Uparrow_{[]} d}\ \text{PPO}_{[]}\text{BoundVar} \qquad \frac{x \notin \sigma}{\sigma, \vdash x \Uparrow_{[]} x}\ \text{PPO}_{[]}\text{UnboundVar}$$

$$\frac{\sigma \vdash d \Uparrow_{[]} d'}{\sigma \vdash \lambda x.d \Uparrow_{[]} \lambda x.d'}\ \text{PPO}_{[]}\text{Lam} \qquad \frac{\sigma \vdash d_1 \Uparrow_{[]} d_1' \qquad \sigma \vdash d_2 \Uparrow_{[]} d_2'}{\sigma \vdash d_1(d_2) \Uparrow_{[]} d_1'(d_2')}\ \text{PPO}_{[]}\text{Ap}$$

$$\frac{}{\sigma \vdash (\!|\!)^u \Uparrow_{[]} [\sigma](\!|\!)^u}\ \text{PPO}_{[]}\text{EHole} \qquad \frac{\sigma \vdash d \Uparrow_{[]} d'}{\sigma \vdash (\!|d|\!)^u \Uparrow_{[]} [\sigma](\!|d'|\!)^u}\ \text{PPO}_{[]}\text{NEHole}$$

Figure: Big-step semantics for substitution postprocessing

# Generalized closures

| Interpretation | Sample expression |
|---|---|
| Function closure | $[\sigma]\lambda x.d$ |
| Hole closure | $[\sigma]⦇d⦈^u$ |
| Closure around unmatched `let` | $[\sigma](\texttt{let}\ x = d_1\ \texttt{in}\ d_2)$ |
| Closure around unmatched `case` | $[\sigma](\texttt{case}\ x\ \texttt{of}\ \text{rules})$ |
| Closure around filled hole | $[\![\sigma]\!]d_{fill}$ |

Table: Examples of generalized closures

# Table of Contents


1. Primer on PL theory
2. The Hazel live programming environment
3. Evaluation using the environment model
4. Identifying hole instances by physical environment
5. The fill-and-resume (FAR) optimization
6. Empirical results
7. Theoretical results/innovations
8. Future work/conclusions

# Motivation for hole instances

```
let a = ⦇⦈¹ in
let b = λ x . { ⦇⦈² } in
f 3 + f 4
```

Figure: Illustration of hole instances

$$[a \leftarrow [\varnothing]⦇⦈^1, x \leftarrow 3]⦇⦈^2 + [a \leftarrow [\varnothing]⦇⦈^1, x \leftarrow 4]⦇⦈^2$$

Figure: Result of Figure 11

# Motivation for hole closures/instantiations I

$\texttt{let a = } (\!|\!)^1 \texttt{ in}$
$\texttt{let b = } (\!|\!)^2 \texttt{ in}$
$\texttt{let c = } (\!|\!)^3 \texttt{ in}$
$\texttt{let d = } (\!|\!)^4 \texttt{ in}$
$\texttt{let e = } (\!|\!)^5 \texttt{ in}$
$\texttt{let f = } (\!|\!)^6 \texttt{ in}$
$\texttt{let g = } (\!|\!)^7 \texttt{ in}$
...
$\texttt{let x = } (\!|\!)^n \texttt{ in}$
$(\!|\!)^{n+1}$

Figure: A Hazel program that generates $2^N$ total hole instances

# Motivation for hole closures/instantiations II

$$\begin{tikzcd}
{[\sigma^4]\llparenthesis\rrparenthesis^4} \arrow[d, "c"'] \arrow[dd, bend left=60, "b"] \arrow[ddd, bend left=80, "a"] \\
{[\sigma^3]\llparenthesis\rrparenthesis^3} \arrow[d, "b"'] \arrow[dd, bend right=60, "a"'] \\
{[\sigma^2]\llparenthesis\rrparenthesis^2} \arrow[d, "a"'] \\
{[\varnothing]\llparenthesis\rrparenthesis^1}
\end{tikzcd}$$

(a) Structure of the result

$$\begin{tikzcd}
 & & & {[\sigma^4]\llparenthesis\rrparenthesis^{4:1}} \arrow[dlll, "a"'] \arrow[dll, "b"] \arrow[d, "c"'] \\
{[\varnothing]\llparenthesis\rrparenthesis^{1:1}} & {[\sigma^2]\llparenthesis\rrparenthesis^{2:1}} \arrow[d, "a"'] & & {[\sigma^3]\llparenthesis\rrparenthesis^{3:1}} \arrow[dl, "a"'] \arrow[d, "b"'] \\
 & {[\varnothing]\llparenthesis\rrparenthesis^{1:2}} & {[\varnothing]\llparenthesis\rrparenthesis^{1:3}} & {[\sigma^2]\llparenthesis\rrparenthesis^{2:2}} \arrow[d, "a"'] \\
 & & & {[\varnothing]\llparenthesis\rrparenthesis^{1:4}}
\end{tikzcd}$$

(b) Numbered hole instances in the result

Figure: Hole numbering in Figure 13

# The hole numbering algorithm

$\boxed{H, p \vdash d \Uparrow_i d' \dashv H'}$ $d$ gets renumbered to $d'$

$$\frac{(u, e) \notin H \qquad i = \mathrm{hid}(H, u) \qquad H, (u, i) \vdash \sigma \Uparrow_i \sigma' \dashv H' \qquad H'' = H, (u, e) \leftarrow (i, \{p\}, \sigma')}{H, p \vdash [\sigma^e](\!|\!)^u \Uparrow_i [\sigma'^e](\!|\!)^{u:i} \dashv H''}\ \mathrm{PP}_i\mathrm{EHoleNew}$$

$$\frac{H = H', (u, e) \leftarrow (i, \{p_i\}, \sigma'^e) \qquad H'' = H, (u, e) \leftarrow (i, \{p_i\} \cup \{p\}, \sigma'^e)}{H, p \vdash [\sigma^e](\!|\!)^u \Uparrow_i [\sigma'^e](\!|\!)^{u:i} \dashv H''}\ \mathrm{PP}_i\mathrm{EHoleFound}$$

$$\frac{\begin{array}{c}(u, e) \notin H \qquad i = \mathrm{hid}(H, u) \\ H, (u, e), \vdash \sigma \Uparrow_i \sigma' \dashv H' \qquad H'' = H, (u, e) \leftarrow (i, \{p\}, \sigma') \qquad H'', p \vdash d \Uparrow_i d' \dashv H'''\end{array}}{H, p \vdash [\sigma^e](\!|d|\!)^u \Uparrow_i [\sigma'^e](\!|d'|\!)^{u:i} \dashv H'''}\ \mathrm{PP}_i\mathrm{NEHoleNew}$$

$$\frac{\begin{array}{c}H = H', (u, e) \leftarrow (i, \{p_i\}, \sigma'^e) \\ H'' = H, (u, e) \leftarrow (i, \{p_i\} \cup \{p\}, \sigma'^e) \qquad H'', p \vdash d \Uparrow_i d' \dashv H'''\end{array}}{H, p \vdash [\sigma^e](\!|d|\!)^u \Uparrow_i [\sigma'^e](\!|d'|\!)^{u:i} \dashv H'''}\ \mathrm{PP}_i\mathrm{NEHoleFound}$$

Figure: Hole closure numbering postprocessing semantics

# A unified postprocessing algorithm

$\boxed{d \Uparrow (H, d')}$ $d$ postprocesses to $d'$ with hole closure info $H$

$$\frac{d \Uparrow_{[]} d' \qquad \varnothing, \varnothing \vdash d' \Uparrow_i d'' \dashv H}{d \Uparrow d'' \dashv H} \text{PP-Result}$$

Figure: Overall postprocessing judgment

# Table of Contents

1 Primer on PL theory

2 The Hazel live programming environment

3 Evaluation using the environment model

4 Identifying hole instances by physical environment

5 The fill-and-resume (FAR) optimization

6 Empirical results

7 Theoretical results/innovations

8 Future work/conclusions

# Motivating example

# The FAR process

# 1-step vs. $n$-step FAR

# Detecting a valid fill operation

# The fill operation

# The resume operation

# The postprocessing operation

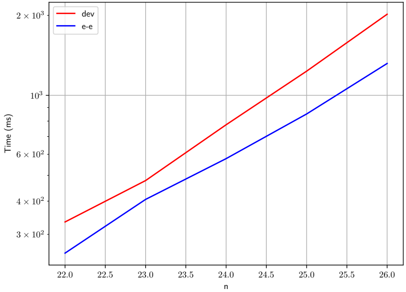
# Table of Contents

1 Primer on PL theory
2 The Hazel live programming environment
3 Evaluation using the environment model
4 Identifying hole instances by physical environment
5 The fill-and-resume (FAR) optimization
6 Empirical results
7 Theoretical results/innovations
8 Future work/conclusions

# Evaluation with environments I

```
let f : Int → Int =
  λ x . {
    case x of
    | 0 ⇒ 0
    | 1 ⇒ 1
    | n ⇒ f (n - 1) + f (n - 2)
    end
  } in
f 25
```

(a) Source



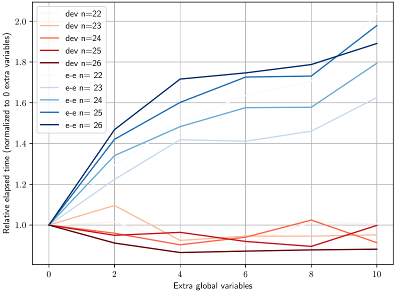

(b) Performance

Figure: A computationally expensive Hazel program with no holes

# Evaluation with environments II

```
let a = 0 in
let b = 0 in
let c = 0 in
let d = 0 in
let e = 0 in
let f : Int → Int =
  λ x . {
    case x of
    | 0 ⇒ 0
    | 1 ⇒ 1
    | n ⇒ f (n - 1) + f (n - 2)
    end
  } in
f 25
```

(a) Source



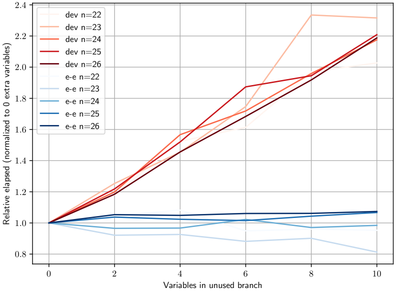

(b) Performance

Figure: Adding global bindings to the fib($n$) program

# Evaluation with environments III

```
let f : Int → Int =
  λ x . {
    case x of
    | 0 ⇒ 0
    | 1 ⇒ 1
    | n ⇒ f (n - 1) + f (n - 2)
    | 0 ⇒ f 0 + f 0 + f 0 + f 0 + f 0
    end
  } in
f 25
```

(a) Source




(b) Performance

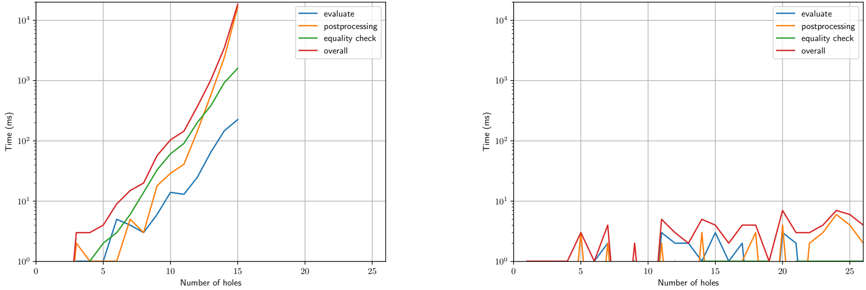
Figure: Adding variable substitutions to unused branches

# Hole numbering motivating example I

$\texttt{let a =}\ \llparenthesis\rrparenthesis^{1}\ \texttt{in}$
$\texttt{let b =}\ \llparenthesis\rrparenthesis^{2}\ \texttt{in}$
$\texttt{let c =}\ \llparenthesis\rrparenthesis^{3}\ \texttt{in}$
$\texttt{let d =}\ \llparenthesis\rrparenthesis^{4}\ \texttt{in}$
$\texttt{let e =}\ \llparenthesis\rrparenthesis^{5}\ \texttt{in}$
$\texttt{let f =}\ \llparenthesis\rrparenthesis^{6}\ \texttt{in}$
$\texttt{let g =}\ \llparenthesis\rrparenthesis^{7}\ \texttt{in}$
$\texttt{...}$
$\texttt{let x =}\ \llparenthesis\rrparenthesis^{n}\ \texttt{in}$
$\llparenthesis\rrparenthesis^{n+1}$

Figure: A Hazel program that generates $2^N$ total hole instances

# Hole numbering motivating example II




(a) `dev` branch

(b) `eval-environment` branch

Figure: Performance of evaluating program in Figure 13

# FAR motivating example I

| Program | Steps | Steps (w/ FAR) | Step Δ | Cumulative Step Δ |
|---|---|---|---|---|
| `let f = ... in`<br>`let a =` ⦇⦈$^1$ `in`<br>⦇⦈$^2$ | 7 | - | 0 | 0 |
| `let f = ... in`<br>`let a = f in`<br>⦇⦈$^2$ | 12 | 21 | 9 | 9 |
| `let f = ... in`<br>`let a = f` ⦇⦈$^3$ `in`<br>⦇⦈$^2$ | 17 | - | 0 | 9 |
| `let f = ... in`<br>`let a = f 2 in`<br>⦇⦈$^2$ | 58 | 69 | 11 | 20 |

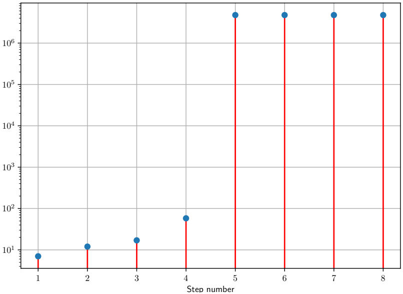
Table: A program edit history with an expensive computation

# FAR motivating example II

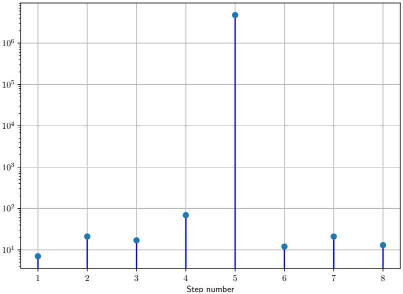
| Program | Steps | Steps (w/ FAR) | Step Δ | Cumulative Step Δ |
|---|---|---|---|---|
| `let f = ... in`<br>`let a = f 25 in`<br>$\llparenthesis\rrparenthesis^{2}$ | 4762964 | - | 0 | 20 |
| `let f = ... in`<br>`let a = f 25 in`<br>$\llparenthesis\rrparenthesis^{2}$ + $\llparenthesis\rrparenthesis^{4}$ | 4762966 | 12 | -4762954 | -4762934 |
| `let f = ... in`<br>`let a = f 25 in`<br>$\llparenthesis\rrparenthesis^{2}$ + 2 | 4762966 | 21 | -4762954 | -9525879 |
| `let f = ... in`<br>`let a = f 25 in`<br>`a + 2` | 4792967 | 13 | -4792954 | -14288813 |

Table: A program edit history with an expensive computation, cont'd.

# FAR motivating example III




(a) Normal evaluation




(b) With one-step FAR

Figure: Number of evaluation steps per edit in Table 2

# Table of Contents


1. Primer on PL theory
2. The Hazel live programming environment
3. Evaluation using the environment model
4. Identifying hole instances by physical environment
5. The fill-and-resume (FAR) optimization
6. Empirical results
7. Theoretical results/innovations
8. Future work/conclusions

# Generalized closures

# Unique hole closures

# FAR as a generalization of evaluation

# Table of Contents


1. Primer on PL theory
2. The Hazel live programming environment
3. Evaluation using the environment model
4. Identifying hole instances by physical environment
5. The fill-and-resume (FAR) optimization
6. Empirical results
7. Theoretical results/innovations
8. Future work/conclusions

# Future work

*n*-step FAR
: Integrate edit history into FAR

Generalized memoization
: Unify notation and metatheory of memoization

Formal evaluation of metatheory
: Check coverage and correctness of metatheorems using Agda

# Conclusions

**Evaluation with environments** Expected performance gains, implementation remains functionally pure

**Generalized closures** Simplify many parts of the implementation, also useful for FAR

**Memoization of environments** Applicable for postprocessing, equality checking, resume operation

**FAR PoC** Including *n*-step detection, re-evaluation of closures

**Plausible metatheory** For future work in Agda

# References I


Jeremy G. Siek and Walid Taha.
Gradual typing for functional languages.
In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pages 81–92, 2006.

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland.
Refined criteria for gradual typing.
In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer.
Hazelnut: A Bidirectionally Typed Structure Editor Calculus.
In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, 2017.

# References II


Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer.
Live functional programming with typed holes.
*PACMPL*, 3(POPL), 2019.