

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

Jonathan Lam<sup>1</sup>   Prof. Fred Fontaine, Advisor<sup>1</sup>  
Prof. Robert Marano, Co-advisor<sup>1</sup>   Prof. Cyrus Omar<sup>2</sup>

<sup>1</sup>Electrical Engineering  
The Cooper Union for the Advancement of Science and Art

<sup>2</sup>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

$$\begin{aligned}\tau &::= \tau \rightarrow \tau \mid b \mid () \\ e &::= c \mid x \mid \lambda x : \tau. e \mid e \ e \mid e : \tau \mid () \mid (e)\end{aligned}$$

Figure: Hazelnut grammar

# Static and dynamic semantics

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

**Dynamics** Evaluation (“run-time”)

$$\frac{e_1 \Downarrow \lambda x. e'_1 \quad e_2 \Downarrow e'_2 \quad [e'_2/x]e'_1 = e}{e_1 \ e_2 \Downarrow e} \text{EAp}$$

**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

$$\begin{array}{l}
 e ::= x \\
 \quad | \lambda x. e \\
 \quad | e \ e
 \end{array}
 \qquad
 \begin{array}{c}
 \frac{}{\lambda x. e \Downarrow \lambda x. e} \Lambda\text{-ELam} \\
 \\
 \frac{e_1 \Downarrow \lambda x. e'_1 \quad [e_2/x]e'_1 \Downarrow e}{e_1 \ e_2 \Downarrow e} \Lambda\text{-EAp}
 \end{array}$$

(a) Grammar

(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

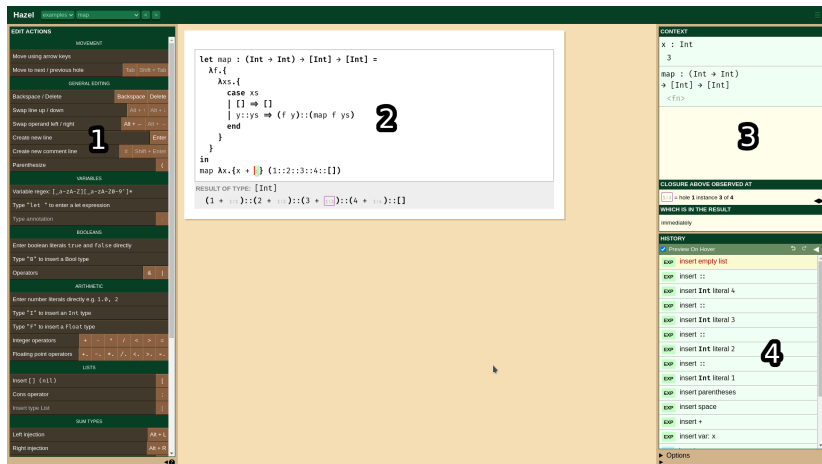


Figure: The Hazel interface

# 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	$\lambda x. \langle \rangle$	construct lam x	(13e)
2	$(\lambda x. \langle \rangle : (\langle \rangle \rightarrow \langle \rangle))$	construct num	(12b)
3	$(\lambda x. \langle \rangle : (\langle \rangle \rightarrow \langle \rangle))$	move parent	(6c)
4	$(\lambda x. \langle \rangle : (\langle \rangle \rightarrow \langle \rangle))$	move child 2	(6b)
5	$(\lambda x. \langle \rangle : (\langle \rangle \rightarrow \langle \rangle))$	construct num	(12b)
6	$(\lambda x. \langle \rangle : (\langle \rangle \rightarrow \langle \rangle))$	move parent	(6d)
7	$(\lambda x. \langle \rangle : (\langle \rangle \rightarrow \langle \rangle))$	move parent	(8d)
8	$(\lambda x. \langle \rangle : (\langle \rangle \rightarrow \langle \rangle))$	move child 1	(8a)
9	$(\lambda x. \langle \rangle : (\langle \rangle \rightarrow \langle \rangle))$	move child 1	(8e)
10	$(\lambda x. \langle \rangle : (\langle \rangle \rightarrow \langle \rangle))$	construct var x	(13c)
11	$(\lambda x. \langle \rangle : (\langle \rangle \rightarrow \langle \rangle))$	construct plus	(13l)
12	$(\lambda x. \langle \rangle : (\langle \rangle \rightarrow \langle \rangle))$	construct lit 1	(13j)
13	$(\lambda x. \langle \rangle : (\langle \rangle \rightarrow \langle \rangle))$	—	—

Figure 1. Constructing the increment function in Hazelnut.

now assume $incr : \text{num} \rightarrow \text{num}$			
#	Z-Expression	Next Action	Rule
14	$\langle \rangle$	construct var $incr$	(13c)
15	$\langle \rangle$	construct ap	(13h)
16	$incr(\langle \rangle)$	construct var $incr$	(13d)
17	$incr(\langle \rangle)$	construct ap	(13h)
18	$incr(incr(\langle \rangle))$	construct lit 3	(13j)
19	$incr(incr(\langle \rangle))$	move parent	(8j)
20	$incr(incr(\langle \rangle))$	move parent	(8p)
21	$incr(incr(\langle \rangle))$	finish	(16b)
22	$incr(incr(\langle \rangle))$	—	—

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

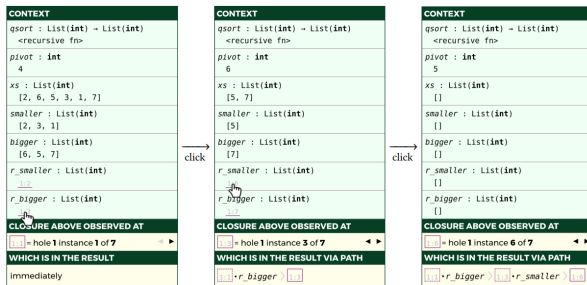


Figure: Illustration of Hazelnut Live context inspector [4]

# 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

$\sigma \vdash d \Downarrow d'$   $d$  evaluates to  $d'$  given environment  $\sigma$

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

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

$$\frac{\sigma \vdash d_1 \Downarrow [\sigma']\lambda x : \tau. d' \quad \sigma \vdash d_2 \Downarrow d'_2 \quad \sigma', x \leftarrow d'_2 \vdash d'_1 \Downarrow d}{\sigma \vdash d_1 d_2 \Downarrow d} \text{EAp}$$

$$\frac{}{\sigma \vdash (d)^u \Downarrow [\sigma](d)^u} \text{EvalB-EHole}$$

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

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

$$\begin{array}{c}
 \frac{\sigma \vdash d \Downarrow [\sigma']d'}{\sigma \vdash \text{fix } f : \tau. d \Downarrow [\sigma, f \leftarrow \text{fix } f : \tau. [\sigma']d']d'} \text{EFix} \\
 \\
 \frac{d \neq \text{fix } f : \tau. d'}{\sigma, x \leftarrow d \vdash x \Downarrow d} \text{EVar} \qquad \frac{\sigma \vdash \text{fix } f : \tau. d \Downarrow d'}{\sigma, x \leftarrow \text{fix } f : \tau. d \vdash x \Downarrow d'} \text{EUnwind}
 \end{array}$$

**Figure:** Big-step semantics for evaluation of fixpoints



# Matching the result from evaluation using substitution

$d \uparrow_{\square} d'$   $d$  is substitutes to  $d'$  inside the evaluation boundary

$$\frac{}{c \uparrow_{\square} c} \text{PPI}_{\square} \text{Const}$$

$$\frac{d_1 \uparrow_{\square} d'_1 \quad d_2 \uparrow_{\square} d'_2}{d_1 \ d_2 \uparrow_{\square} d'_1 \ d'_2} \text{PPI}_{\square} \text{Ap}$$

$$\frac{\sigma \uparrow_{\square} \sigma' \quad \sigma' \vdash d \uparrow_{\square} d'}{[\sigma]d \uparrow_{\square} d'} \text{PPI}_{\square} \text{Closure}$$

$\sigma \vdash d \uparrow_{\square} d'$   $d$  substitutes to  $d'$  outside the evaluation boundary

$$\frac{}{c \uparrow_{\square} c} \text{PPO}_{\square} \text{Const}$$

$$\frac{}{\sigma, x \leftarrow d \vdash x \uparrow_{\square} d} \text{PPO}_{\square} \text{BoundVar}$$

$$\frac{x \notin \sigma}{\sigma, \vdash x \uparrow_{\square} x} \text{PPO}_{\square} \text{UnboundVar}$$

$$\frac{\sigma \vdash d \uparrow_{\square} d'}{\sigma \vdash \lambda x. d \uparrow_{\square} \lambda x. d'} \text{PPO}_{\square} \text{Lam}$$

$$\frac{\sigma \vdash d_1 \uparrow_{\square} d'_1 \quad \sigma \vdash d_2 \uparrow_{\square} d'_2}{\sigma \vdash d_1(d_2) \uparrow_{\square} d'_1(d'_2)} \text{PPO}_{\square} \text{Ap}$$

$$\frac{}{\sigma \vdash \langle d \rangle^u \uparrow_{\square} [\sigma] \langle d \rangle^u} \text{PPO}_{\square} \text{EHole}$$

$$\frac{\sigma \vdash d \uparrow_{\square} d'}{\sigma \vdash \langle d \rangle^u \uparrow_{\square} [\sigma] \langle d \rangle^u} \text{PPO}_{\square} \text{NEHole}$$

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 <code>let</code>	$[\sigma](\text{let } x = d_1 \text{ in } d_2)$
Closure around unmatched <code>case</code>	$[\sigma](\text{case } x \text{ of rules})$
Closure around filled hole	$\llbracket \sigma \rrbracket d_{fill}$

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 =  $\bigcirc^1$  in
let b =  $\lambda x . \{ \bigcirc^2 \}$  in
f 3 + f 4

```

Figure: Illustration of hole instances

$$[a \leftarrow [\emptyset] \bigcirc^1, x \leftarrow 3] \bigcirc^2 + [a \leftarrow [\emptyset] \bigcirc^1, x \leftarrow 4] \bigcirc^2$$

Figure: Result of Figure 11

# Motivation for hole closures/instantiations I

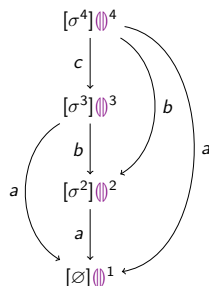
```

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

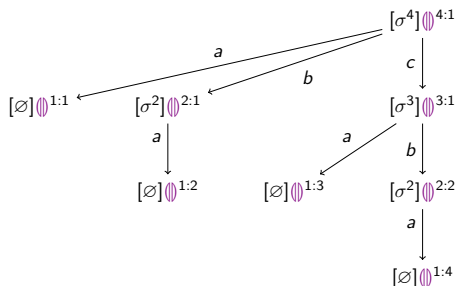
```

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

# Motivation for hole closures/instantiations II



(a) Structure of the result



(b) Numbered hole instances in the result

Figure: Hole numbering in Figure 13

# The hole numbering algorithm

$H, p \vdash d \uparrow_i d' \dashv H'$   $d$  gets renumbered to  $d'$

$$\frac{(u, e) \notin H \quad i = \text{hid}(H, u) \quad H, (u, i) \vdash \sigma \uparrow_i \sigma' \dashv H' \quad H'' = H, (u, e) \leftarrow (i, \{p\}, \sigma'^e)}{H, p \vdash [\sigma^e] \textcircled{u} \uparrow_i [\sigma'^e] \textcircled{u:i} \dashv H''} \text{PP}_i\text{EHoleNew}$$

$$\frac{H = H', (u, e) \leftarrow (i, \{p_i\}, \sigma'^e) \quad H'' = H, (u, e) \leftarrow (i, \{p_i\} \cup \{p\}, \sigma'^e)}{H, p \vdash [\sigma^e] \textcircled{u} \uparrow_i [\sigma'^e] \textcircled{u:i} \dashv H''} \text{PP}_i\text{EHoleFound}$$

$$\frac{\begin{array}{c} (u, e) \notin H \quad i = \text{hid}(H, u) \\ H, (u, e) \vdash \sigma \uparrow_i \sigma' \dashv H' \quad H'' = H, (u, e) \leftarrow (i, \{p\}, \sigma') \quad H'', p \vdash d \uparrow_i d' \dashv H''' \end{array}}{H, p \vdash [\sigma^e] \textcircled{d} \uparrow_i [\sigma'^e] \textcircled{d'} \textcircled{u:i} \dashv H'''} \text{PP}_i\text{NEHoleNew}$$

$$\frac{H = H', (u, e) \leftarrow (i, \{p_i\}, \sigma'^e) \quad H'' = H, (u, e) \leftarrow (i, \{p_i\} \cup \{p\}, \sigma'^e) \quad H'', p \vdash d \uparrow_i d' \dashv H'''}{H, p \vdash [\sigma^e] \textcircled{d} \uparrow_i [\sigma'^e] \textcircled{d'} \textcircled{u:i} \dashv H'''} \text{PP}_i\text{NEHoleFound}$$

Figure: Hole closure numbering postprocessing semantics

# A unified postprocessing algorithm

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

$$\frac{d \uparrow_{\square} d' \quad \emptyset, \emptyset \vdash d' \uparrow_i d'' \dashv H}{d \uparrow d'' \dashv H} \text{ PP-Result}$$

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

# Hole numbering motivating example

# FAR motivating example

# 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.