

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

Jonathan Lam¹ Prof. Fred Fontaine, Advisor¹
Prof. Robert Marano, Co-advisor¹ Prof. Cyrus Omar²

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

²Electrical 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 Discussion, future work, and 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”)

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

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

A brief primer on the λ -calculus

- Untyped λ -calculus Simple universal model of computation by Church
 Simply-typed λ -calculus Extension of the ULC with static type-checking
 Gradually-typed λ -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 λ -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 Discussion, future work, and 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

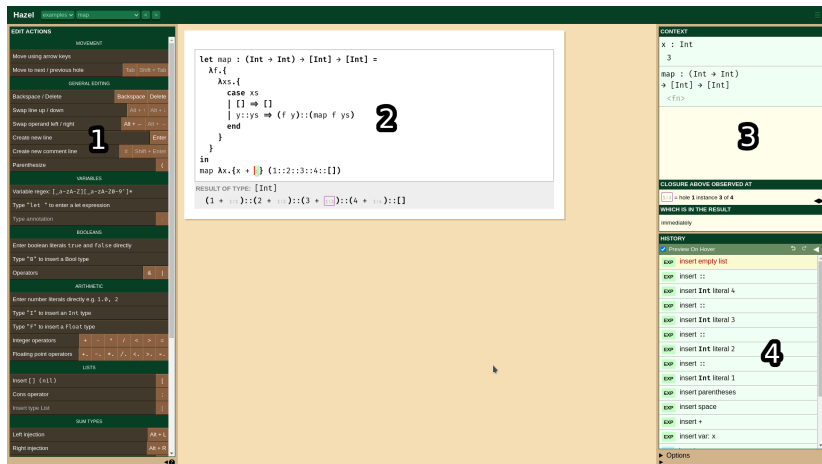


Figure: The Hazel interface

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

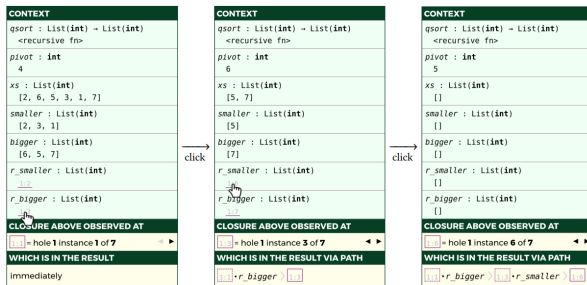


Figure: Illustration of Hazelnut Live context inspector [4]

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 Discussion, future work, and conclusions

Evaluation using environments vs. substitution

```
let x = 3 in
  if True then 0 else x
```

(a) Expression with variable binding

if True then 0 else 3 $\{x \leftarrow 3\} \vdash (\text{if True then 0 else } x)$

(b) Substitution (eager)

(c) Environments (lazy)

Figure: Comparison of variable binding methods

Updated evaluation rules

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

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

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

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

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

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

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

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

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

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

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

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

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

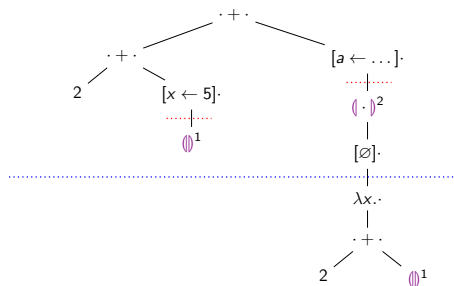
The evaluation boundary

```
let a = λ x . { 2 + (a)2 } in
a 5 + (a)2
```

(a) Program

$$\Downarrow (2 + [x \leftarrow 5](a)^1) + [a \leftarrow \dots](\lambda x. 2 + (a)^1)^2$$

(b) Program result



(c) Program result AST

Figure: Illustration of evaluation boundary

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 Discussion, future work, and conclusions

Motivation for hole instances

```
let a =  $\emptyset^1$  in
let f =  $\lambda x . \{ \emptyset^2 \}$  in
f 3 + f 4
```

Figure: Illustration of hole instances

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

Figure: Result of Figure 13

Motivation for hole closures/instantiations I

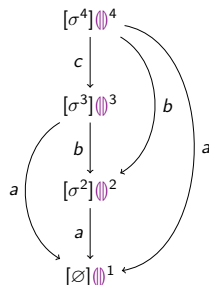
```

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
 $\bigoplus^{n+1}$ 

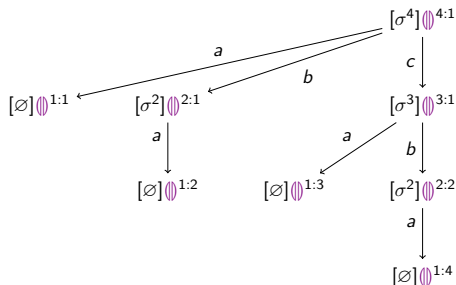
```

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 15

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 Discussion, future work, and conclusions

Motivating example I

What happens if we want to fill the hole hole^1 with the expression $x + 2$?

```
let f : Int → Int =
  λ x . {
    case x of
    | 0 ⇒ 0
    | 1 ⇒ 1
    | n ⇒ f (n - 1) + f (n - 2)
    end
  }
in x = f 30
in  $\text{hole}^1$ 
```

Figure: A sample program with an expensive calculation

Motivating example II

$$[f \leftarrow [\emptyset]\lambda x.\{\dots\}, x \leftarrow 832040] \parallel^1$$

Figure: Result of expensive calculation

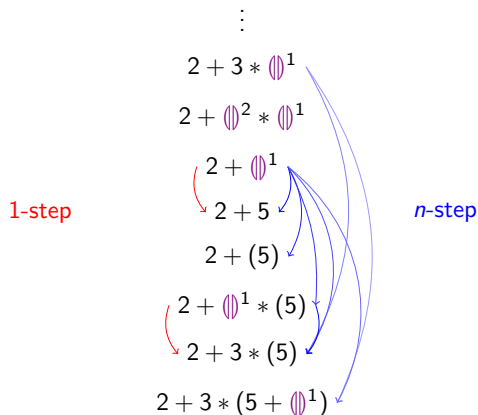
$$\begin{aligned} [f \leftarrow [\emptyset]\lambda x.\{\dots\}, x \leftarrow 832040](x + 2) \\ 832040 + 2 \\ 832042 \end{aligned}$$

Figure: Fill and resume

The FAR process

Check if a fill is appropriate. If not, evaluate as usual. If so, then:

- 1 Detect fill parameters (u , d)
- 2 “Fill”: substitute d for every instance of u
- 3 “Resume”: resume evaluation

1-step vs. n -step FARFigure: 1-step vs. n -step FAR detection

Detecting a valid fill operation

Structural diff algorithm Intuitive, fast n -step FAR detection;
find the smallest hole that subsumes the diff root

$$\lambda x. (\text{hole})^3 \longrightarrow \lambda x. 4$$

$$u = 3$$

$$d = 4$$

$$2 + (\lambda x. 3)^1 \longrightarrow 2 + 5 * (\text{hole})^1$$

$$u = 1$$

$$d = 5 * (\text{hole})^1$$

The fill and resume operations

The fill operation

- Mark closures un-final

$$\llbracket [\sigma] d / [\sigma] d \rrbracket d_{result}$$
- Fill hole instances

$$[d_{fill} / \text{hole}^{u_{fill}}] d_{result}$$

The resume operation

- Evaluate as normal, except:
- Re-evaluate closures

$$\llbracket [\sigma] d \rrbracket \Downarrow [\sigma'] d'$$

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 Discussion, future work, and conclusions

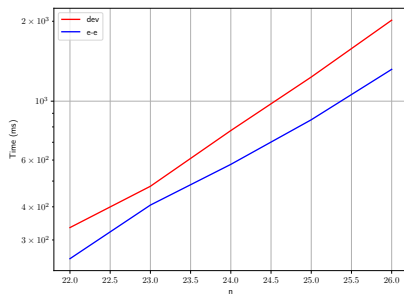
Evaluation with environments I

```

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

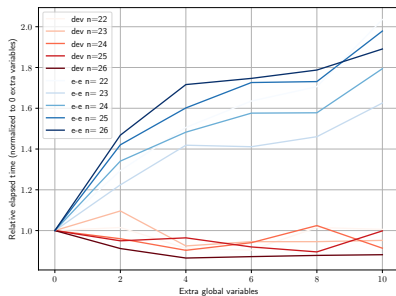
Evaluation with environments II

```

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)
  } in
f 25

```

(a) Source



(b) Performance

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

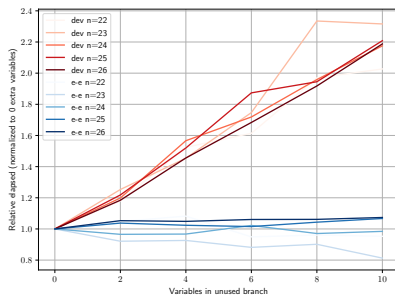
Evaluation with environments III

```

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

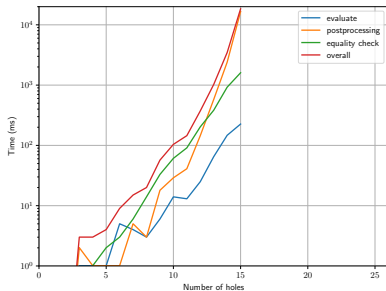
```

let a = (0)1 in
let b = (0)2 in
let c = (0)3 in
let d = (0)4 in
let e = (0)5 in
let f = (0)6 in
let g = (0)7 in
...
let x = (0)n in
(0)n+1

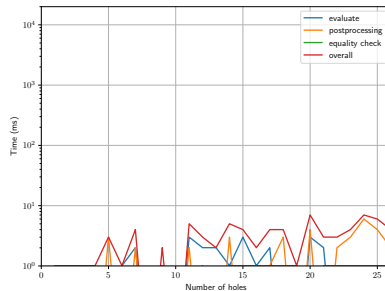
```

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 15

FAR motivating example I

Program	Steps	Steps (w/ FAR)	Step Δ	Cumulative Step Δ
<pre>let f = ... in let a = $\textcircled{1}$ in $\textcircled{1}^2$</pre>	7	-	0	0
<pre>let f = ... in let a = f in $\textcircled{1}^2$</pre>	12	21	9	9
<pre>let f = ... in let a = f $\textcircled{1}^3$ in $\textcircled{1}^2$</pre>	17	-	0	9
<pre>let f = ... in let a = f $\textcircled{1}^2$ in $\textcircled{1}^2$</pre>	58	69	11	20

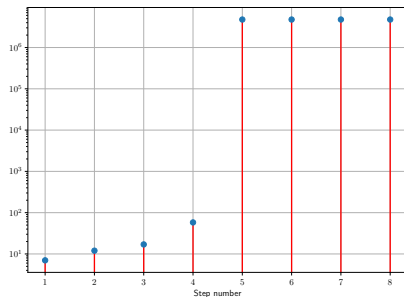
Table: A program edit history with an expensive computation

FAR motivating example II

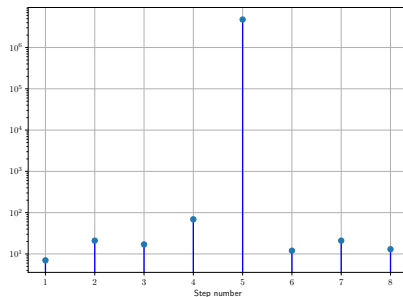
Program	Steps	Steps (w/ FAR)	Step Δ	Cumulative Step Δ
<pre>let f = ... in let a = f 25 in (\emptyset)²</pre>	4762964	-	0	20
<pre>let f = ... in let a = f 25 in (\emptyset)² + (\emptyset)⁴</pre>	4762966	12	-4762954	-4762934
<pre>let f = ... in let a = f 25 in (\emptyset)² + 2</pre>	4762966	21	-4762954	-9525879
<pre>let f = ... in let a = f 25 in a + 2</pre>	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 Discussion, future work, and conclusions

Innovations of this work

Generalized closures Useful for evaluation and memoization

Unique hole closures Grouping hole instances by environment

FAR as a generalization of evaluation Each edit is a n -step FAR

Proposed updates to the evaluation model I

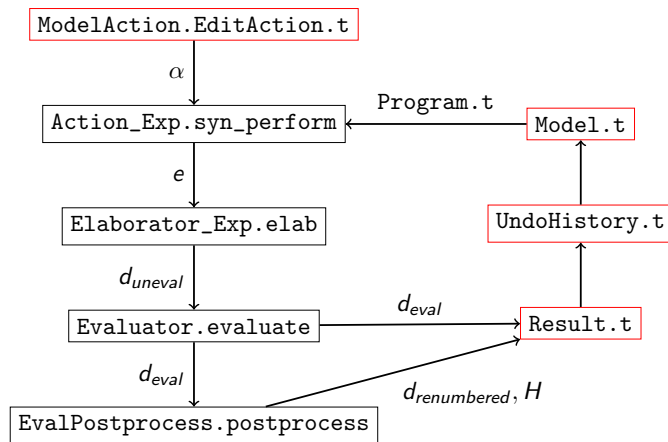


Figure: Previous evaluation model

Proposed updates to the evaluation model II

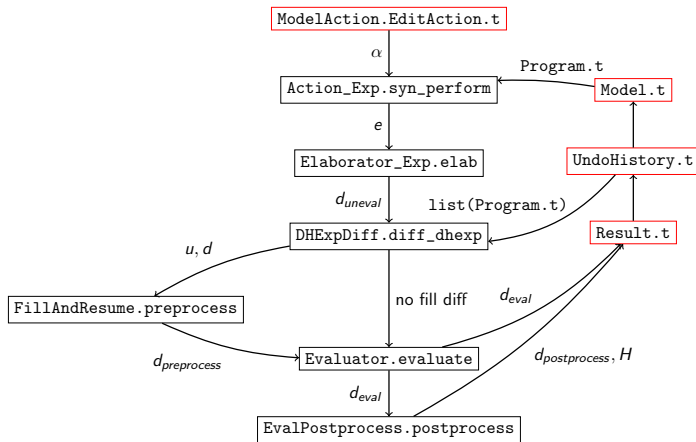


Figure: Proposed evaluation model

Future work

Fully automatic FAR Integrate FAR into the Hazel MVC model

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

User editing studies Gather data on “true” performance impact

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.



Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer.

Live functional programming with typed holes.

PACMPL, 3(POPL), 2019.