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

Project context

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

Functional programming Context for PL theory

Implementation-based Mostly practically-driven

Project goal

Improve aspects of Hazel evaluation Mostly performance-related

Overview

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

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")

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

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

$$e ::= x \mid \lambda x.e \mid e\ e$$

(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 λ-calculus

Table of Contents

The Hazel programming language and environment

Live programming Rapid static and dynamic feedback ("gap problem")

Structured editor Elimination of syntax errors

Gradually typed Hole type and cast-calculus based on GTLC

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

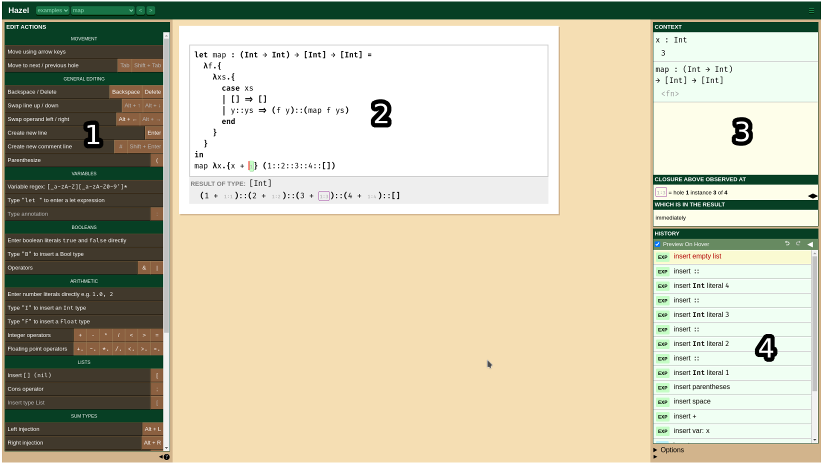
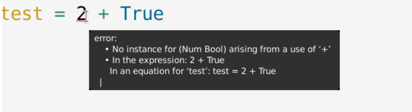


Figure: The Hazel interface

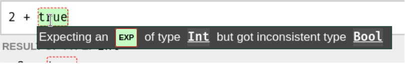
Hazelnut: A bidirectionally-typed static semantics

(Typed) expression holes Internalize “red squiggly underlines” [1]

Action semantics Structural editing behavior, ensures always well-typed



(a) Haskell static type error



(b) Hazel non-empty hole

Figure: “Red squiggly underline”

Hazelnut Live: A bidirectionally-typed dynamic semantics

Internal language Cast calculus from GTLC for dynamic typing

Hole evaluation Evaluation continues *around* holes, captures environment

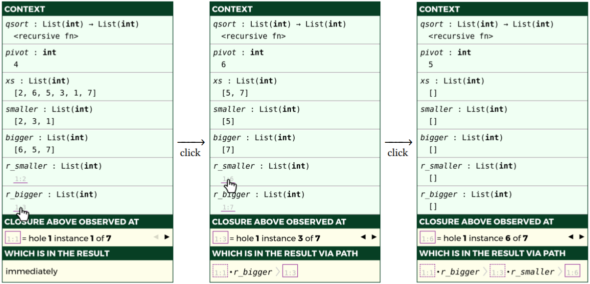


Figure: Illustration of Hazelnut Live context inspector [2]

Table of Contents

Evaluation using environments vs. substitution

$$\texttt{let } x = 3 \texttt{ in}$$
$$\texttt{if True then 0 else } x$$

(a) Expression with variable binding

$$\texttt{if True then 0 else 3}$$

(b) Substitution (eager)

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

(c) Environments (lazy)

Figure: Comparison of variable binding methods

Updated evaluation rules

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

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

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

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

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

Figure: Big-step semantics for evaluation with environments

Matching the result from evaluation using substitution

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

$$\frac{\sigma \Uparrow_{[]} \sigma' \qquad \sigma' \vdash d \Uparrow_{[]} d'}{[\sigma]d \Uparrow_{[]} d'} \text{ PPI}_{[]}\text{Closure}$$

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

$$\frac{}{\sigma, x \leftarrow d \vdash x \Uparrow_{[]} d} \text{ PPO}_{[]}\text{BoundVar}$$

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

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

Figure: Big-step semantics for substitution postprocessing

Generalized closures

Notation in blue is non-standard

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`	$[\sigma](\texttt{case } x \texttt{ of rules})$
Closure around filled hole	$[\![\sigma]\!] d_{fill}$

Table: Examples of generalized closures

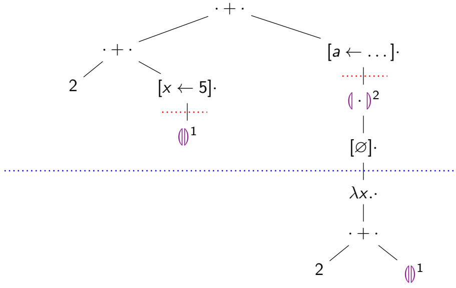
The evaluation boundary

```
let a = λ x . { 2 + (||)^1 } in
a 5 + (|a|)^2
```

(a) Program

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

(b) Program result



(c) Program result AST

Figure: Illustration of evaluation boundary

Table of Contents

Motivation for hole instances

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

Figure: Illustration of hole instances

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

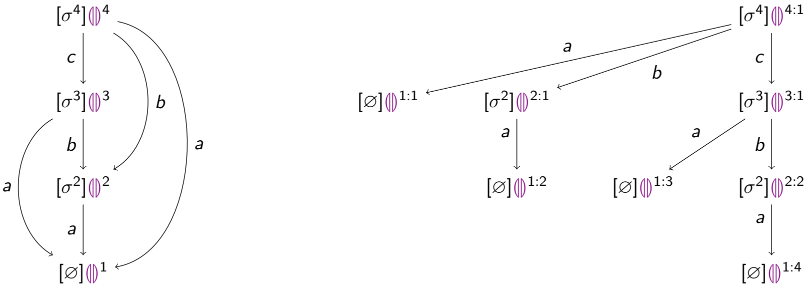
Figure: Result of Figure 12

Motivation for hole closures/instantiations

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

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

Motivation for hole closures/instantiations



(a) Structure of the result (b) Numbered hole instances in the result

Figure: Hole numbering in Figure 14

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

Motivating example

What happens if we want to fill the 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 (||)¹
```

Figure: A sample program with an expensive calculation

Motivating example

$$[f \leftarrow [\varnothing]\lambda x.\{\dots\}, x \leftarrow 832040](\!|\,|\!)^1$$

Figure: Result of expensive calculation

$$[f \leftarrow [\varnothing]\lambda x.\{\dots\}, x \leftarrow 832040](x+2)$$
$$832040 + 2$$
$$832042$$

Figure: Fill and resume

The FAR process

Check if a fill is appropriate. If so, then:

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

If not, evaluate as usual.

1-step vs. *n*-step FAR

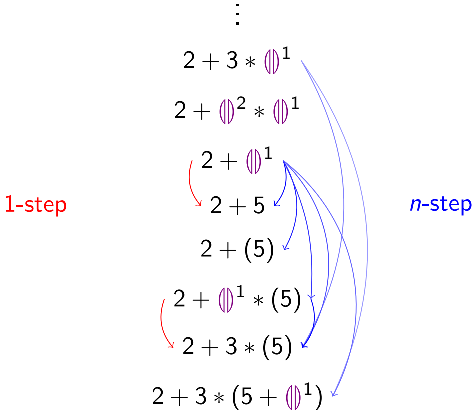


Figure: 1-step vs. *n*-step FAR detection

Detecting a valid fill operation

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

$$\lambda x.\llparenthesis\rrparenthesis^3 \quad \longrightarrow \quad \lambda x.4$$

$$u = 3$$

$$d = 4$$

$$2 + \llparenthesis \lambda x.3 \rrparenthesis^1 \quad \longrightarrow \quad 2 + 5 * \llparenthesis\rrparenthesis^1$$

$$u = 1$$

$$d = 5 * \llparenthesis\rrparenthesis^1$$

The fill and resume operations

The fill operation

- Mark closures un-final
  $[\llbracket\sigma\rrbracket d / [\sigma] d] d_{result}$
- Fill hole instances
  $[d_{fill} / (\!|\!)^{u_{fill}}] d_{result}$

The resume operation

- Evaluate as normal, except:
- Re-evaluate closures
  $\llbracket\sigma\rrbracket d \Downarrow [\sigma'] d'$

Proposed updates to the evaluation model

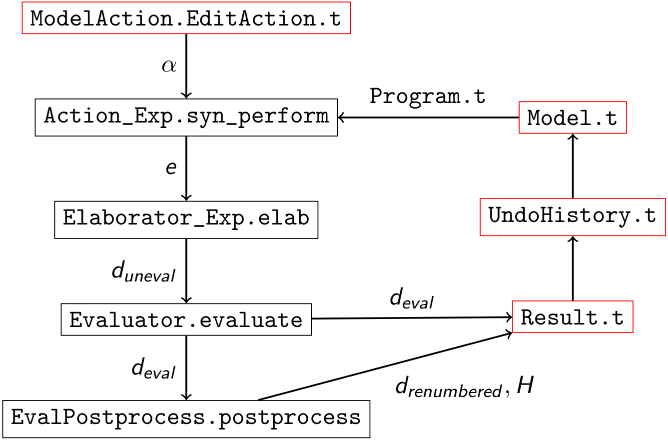


Figure: Previous evaluation model

Proposed updates to the evaluation model

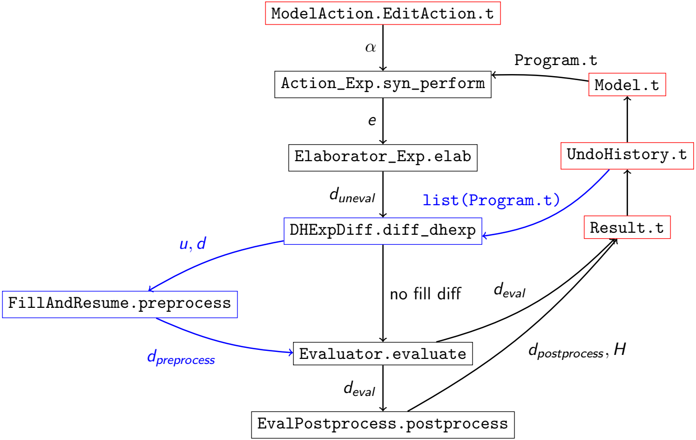


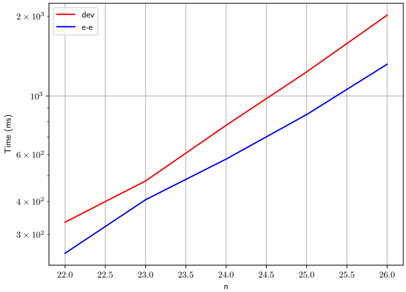
Figure: Proposed evaluation model

Table of Contents

Evaluation with environments

```
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



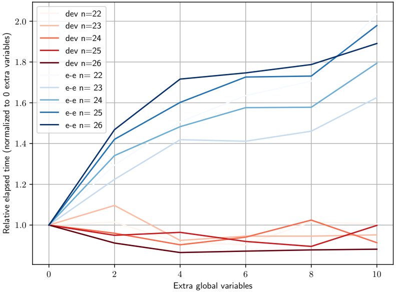
(b) Performance

Figure: A computationally expensive Hazel program with no holes

Evaluation with environments

```
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



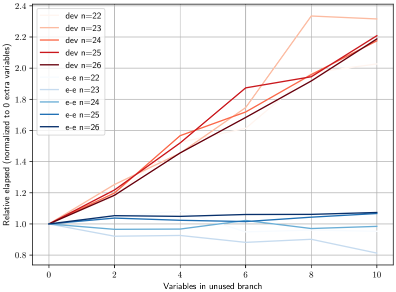
(b) Performance

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

Evaluation with environments

```
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

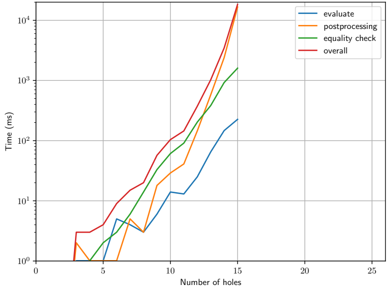
Figure: Adding variable substitutions to unused branches

Hole numbering motivating example

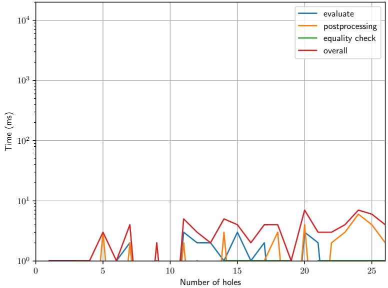
```
let a = ⦇⦈^1 in
let b = ⦇⦈^2 in
let c = ⦇⦈^3 in
let d = ⦇⦈^4 in
let e = ⦇⦈^5 in
let f = ⦇⦈^6 in
let g = ⦇⦈^7 in
...
let x = ⦇⦈^n in
⦇⦈^(n+1)
```

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

Hole numbering motivating example



(a) `dev` branch



(b) `eval-environment` branch

Figure: Performance of evaluating program in Figure 14

FAR motivating example

Program	Steps	Steps (w/ FAR)	Step Δ	Cumulative Step Δ

`let f = ... in`
`let a = ⦇⦈`^{1} `in`
⦇⦈^{2}
	7	-	0	0

`let f = ... in`
`let a = f in`
⦇⦈^{2}
	12	21	9	9

`let f = ... in`
`let a = f ⦇⦈`^{3} `in`
⦇⦈^{2}
	17	-	0	9

`let f = ... in`
`let a = f 2 in`
⦇⦈^{2}
	58	69	11	20

Table: A program edit history with an expensive computation

FAR motivating example

Program	Steps	Steps (w/ FAR)	Step Δ	Cumulative Step Δ

`let f = ... in`
`let a = f 25 in`
⦇⦈^{2}
	4762964	-	0	20

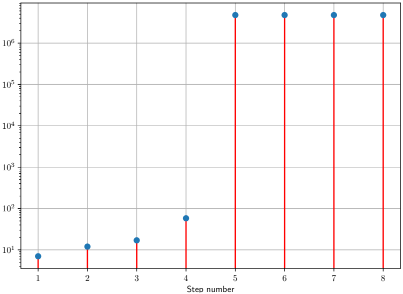
`let f = ... in`
`let a = f 25 in`
⦇⦈^{2} + ⦇⦈^{4}
	4762966	12	-4762954	-4762934

`let f = ... in`
`let a = f 25 in`
⦇⦈^{2} + 2
	4762966	21	-4762954	-9525879

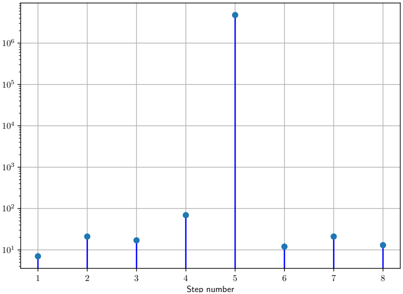
`let f = ... in`
`let a = f 25 in`
`a + 2`
	4792967	13	-4792954	-14288813

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

FAR motivating example



(a) Normal evaluation



(b) With one-step FAR

Figure: Number of evaluation steps per edit in Table 2

Table of Contents

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

Metatheory

Invariants of the evaluation steps; informally justified

- Preservation
- Evaluation boundary
- Singular evaluation boundary
- Substitution postprocessing closures
- Evaluation with environments correctness
- Hole numbering postprocessing
- Fill operation
- Resume operation

Future work

Fully automatic FAR
: Integrate FAR into the Hazel MVC model

n-step FAR
: Integrate edit history into FAR

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

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.